

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

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

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

Order Number: 325383-050US  
February 2014

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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

The products described in this document 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® AES-NI requires a computer system with an AES-NI enabled processor, as well as non-Intel software to execute the instructions in the correct sequence. AES-NI is available on select Intel® processors. For availability, consult your reseller or system manufacturer. For more information, see <http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/>.

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

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

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

Enabling Execute Disable Bit functionality requires a PC with a processor with Execute Disable Bit capability and a supporting operating system. Check with your PC manufacturer on whether your system delivers Execute Disable Bit functionality.

Intel, the Intel logo, Pentium, Xeon, Intel NetBurst, Intel Core, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo, Intel Core 2 Extreme, Intel Pentium D, Itanium, Intel SpeedStep, MMX, Intel Atom, and VTune are trademarks of Intel Corporation in the U.S. and/or other countries.

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

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

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

Copyright © 1997-2014 Intel Corporation. All rights reserved.

# CONTENTS

	PAGE
<b>CHAPTER 1</b>	
<b>ABOUT THIS MANUAL</b>	
1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL .....	1-1
1.2 OVERVIEW OF VOLUME 2A, 2B AND 2C: INSTRUCTION SET REFERENCE .....	1-3
1.3 NOTATIONAL CONVENTIONS .....	1-3
1.3.1 Bit and Byte Order .....	1-3
1.3.2 Reserved Bits and Software Compatibility .....	1-4
1.3.3 Instruction Operands .....	1-4
1.3.4 Hexadecimal and Binary Numbers .....	1-5
1.3.5 Segmented Addressing .....	1-5
1.3.6 Exceptions .....	1-5
1.3.7 A New Syntax for CPUID, CR, and MSR Values .....	1-5
1.4 RELATED LITERATURE .....	1-6
<b>CHAPTER 2</b>	
<b>INSTRUCTION FORMAT</b>	
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 .....	2-6
2.2.1 REX Prefixes .....	2-7
2.2.1.1 Encoding .....	2-7
2.2.1.2 More on REX Prefix Fields .....	2-7
2.2.1.3 Displacement .....	2-10
2.2.1.4 Direct Memory-Offset MOVs .....	2-10
2.2.1.5 Immediates .....	2-10
2.2.1.6 RIP-Relative Addressing .....	2-11
2.2.1.7 Default 64-Bit Operand Size .....	2-11
2.2.2 Additional Encodings for Control and Debug Registers .....	2-11
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 .....	2-12
2.3.3 VEX and the 66H, F2H, and F3H prefixes .....	2-12
2.3.4 VEX and the REX prefix .....	2-12
2.3.5 The VEX Prefix .....	2-13
2.3.5.1 VEX Byte 0, bits[7:0] .....	2-14
2.3.5.2 VEX Byte 1, bit [7] - 'R' .....	2-14
2.3.5.3 3-byte VEX byte 1, bit[6] - 'X' .....	2-15
2.3.5.4 3-byte VEX byte 1, bit[5] - 'B' .....	2-15
2.3.5.5 3-byte VEX byte 2, bit[7] - 'W' .....	2-15
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 .....	2-15
2.3.6 Instruction Operand Encoding and VEX.vvvv, ModR/M .....	2-16
2.3.6.1 3-byte VEX byte 1, bits[4:0] - "m-mmmm" .....	2-17
2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- "L" .....	2-17
2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- "pp" .....	2-17
2.3.7 The Opcode Byte .....	2-18
2.3.8 The MODRM, SIB, and Displacement Bytes .....	2-18
2.3.9 The Third Source Operand (Immediate Byte) .....	2-18
2.3.10 AVX Instructions and the Upper 128-bits of YMM registers .....	2-18
2.3.10.1 Vector Length Transition and Programming Considerations .....	2-18

## CONTENTS

	PAGE
2.3.11 AVX Instruction Length .....	2-19
2.3.12 Vector SIB (VSIB) Memory Addressing .....	2-19
2.3.12.1 64-bit Mode VSIB Memory Addressing .....	2-20
2.4 INSTRUCTION EXCEPTION SPECIFICATION.....	2-20
2.4.1 Exceptions Type 1 (Aligned memory reference) .....	2-25
2.4.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned) .....	2-26
2.4.3 Exceptions Type 3 (<16 Byte memory argument) .....	2-27
2.4.4 Exceptions Type 4 (>=16 Byte mem arg no alignment, no floating-point exceptions) .....	2-28
2.4.5 Exceptions Type 5 (<16 Byte mem arg and no FP exceptions) .....	2-29
2.4.6 Exceptions Type 6 (VEX-Encoded Instructions Without Legacy SSE Analogues) .....	2-30
2.4.7 Exceptions Type 7 (No FP exceptions, no memory arg) .....	2-30
2.4.8 Exceptions Type 8 (AVX and no memory argument) .....	2-31
2.4.9 Exception Type 11 (VEX-only, mem arg no AC, floating-point exceptions) .....	2-32
2.4.10 Exception Type 12 (VEX-only, VSIB mem arg, no AC, no floating-point exceptions) .....	2-33
2.5 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS.....	2-33
2.5.1 Exception Conditions for VEX-Encoded GPR Instructions .....	2-34

## CHAPTER 3

### INSTRUCTION SET REFERENCE, A-M

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 .....	3-4
3.1.1.4 Operand Encoding Column in the Instruction Summary Table .....	3-7
3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table .....	3-7
3.1.1.6 CPUID Support Column in the Instruction Summary Table .....	3-7
3.1.1.7 Description Column in the Instruction Summary Table .....	3-7
3.1.1.8 Description Section .....	3-7
3.1.1.9 Operation Section .....	3-8
3.1.1.10 Intel® C/C++ Compiler Intrinsic Equivalents Section.....	3-11
3.1.1.11 Flags Affected Section .....	3-13
3.1.1.12 FPU Flags Affected Section .....	3-13
3.1.1.13 Protected Mode Exceptions Section .....	3-13
3.1.1.14 Real-Address Mode Exceptions Section .....	3-14
3.1.1.15 Virtual-8086 Mode Exceptions Section .....	3-14
3.1.1.16 Floating-Point Exceptions Section .....	3-14
3.1.1.17 SIMD Floating-Point Exceptions Section .....	3-15
3.1.1.18 Compatibility Mode Exceptions Section .....	3-15
3.1.1.19 64-Bit Mode Exceptions Section .....	3-15
3.2 INSTRUCTIONS (A-M) .....	3-15
AAA—ASCII Adjust After Addition .....	3-16
AAD—ASCII Adjust AX Before Division .....	3-18
AAM—ASCII Adjust AX After Multiply .....	3-20
AAS—ASCII Adjust AL After Subtraction .....	3-22
ADC—Add with Carry .....	3-24
ADD—Add .....	3-27
ADDPD—Add Packed Double-Precision Floating-Point Values .....	3-29
ADDPS—Add Packed Single-Precision Floating-Point Values .....	3-31
ADDSD—Add Scalar Double-Precision Floating-Point Values .....	3-33
ADDSS—Add Scalar Single-Precision Floating-Point Values .....	3-34
ADDSUBPD—Packed Double-FP Add/Subtract .....	3-35
ADDSUBPS—Packed Single-FP Add/Subtract .....	3-37
AESDEC—Perform One Round of an AES Decryption Flow .....	3-40
AESDECLAST—Perform Last Round of an AES Decryption Flow .....	3-42
AESENC—Perform One Round of an AES Encryption Flow .....	3-44
AESENCLAST—Perform Last Round of an AES Encryption Flow .....	3-46
AESIMC—Perform the AES InvMixColumn Transformation .....	3-48
AESKEYGENASSIST—AES Round Key Generation Assist .....	3-49

	PAGE
AND—Logical AND .....	3-51
ANDN — Logical AND NOT .....	3-53
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values .....	3-54
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values .....	3-56
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values .....	3-58
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values .....	3-60
ARPL—Adjust RPL Field of Segment Selector .....	3-62
BLENDPD — Blend Packed Double Precision Floating-Point Values.....	3-64
BEXTR — Bit Field Extract .....	3-66
BLENDPS — Blend Packed Single Precision Floating-Point Values.....	3-68
BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values.....	3-70
BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values.....	3-72
BLSI — Extract Lowest Set Isolated Bit .....	3-75
BLSMSK — Get Mask Up to Lowest Set Bit .....	3-76
BLSR — Reset Lowest Set Bit .....	3-77
BOUND—Check Array Index Against Bounds .....	3-78
BSF—Bit Scan Forward .....	3-80
BSR—Bit Scan Reverse .....	3-82
BSWAP—Byte Swap .....	3-84
BT—Bit Test .....	3-85
BTC—Bit Test and Complement .....	3-87
BTR—Bit Test and Reset.....	3-89
BTS—Bit Test and Set .....	3-91
BZHI — Zero High Bits Starting with Specified Bit Position.....	3-93
CALL—Call Procedure .....	3-95
CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword.....	3-108
CLC—Clear Carry Flag .....	3-109
CLD—Clear Direction Flag .....	3-110
CLFLUSH—Flush Cache Line .....	3-111
CLI — Clear Interrupt Flag .....	3-113
CLTS—Clear Task-Switched Flag in CRO.....	3-115
CMC—Complement Carry Flag .....	3-116
CMOVcc—Conditional Move .....	3-117
CMP—Compare Two Operands .....	3-121
CMPPD—Compare Packed Double-Precision Floating-Point Values.....	3-123
CMPPS—Compare Packed Single-Precision Floating-Point Values .....	3-130
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands.....	3-136
CMPSD—Compare Scalar Double-Precision Floating-Point Values .....	3-140
CMPSS—Compare Scalar Single-Precision Floating-Point Values .....	3-144
CMPXCHG—Compare and Exchange.....	3-148
CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes .....	3-150
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS .....	3-153
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS .....	3-155
CPUID—CPU Identification.....	3-157
CRC32 — Accumulate CRC32 Value.....	3-188
CVTDQ2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values .....	3-191
CVTDQ2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values .....	3-193
CVTPD2DQ—Convert Packed Double-Precision FP Values to Packed Dword Integers .....	3-195
CVTPD2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers .....	3-197
CVTPD2PS—Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values .....	3-198
CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values .....	3-200
CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values .....	3-201
CVTPS2DQ—Convert Packed Single-Precision FP Values to Packed Dword Integers .....	3-202
CVTPS2PD—Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values .....	3-204
CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers .....	3-206
CVTSD2SI—Convert Scalar Double-Precision FP Value to Integer .....	3-207
CVTSD2SS—Convert Scalar Double-Precision FP Value to Scalar Single-Precision FP Value.....	3-209
CVTSI2SD—Convert Dword Integer to Scalar Double-Precision FP Value .....	3-211
CVTSI2SS—Convert Dword Integer to Scalar Single-Precision FP Value .....	3-213

## CONTENTS

	PAGE
CVTSS2SD—Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value .....	3-215
CVTSS2SI—Convert Scalar Single-Precision FP Value to Dword Integer .....	3-216
CVTPD2DQ—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers .....	3-218
CVTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers .....	3-220
CVTPS2DQ—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers .....	3-221
CVTPS2PI—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers .....	3-223
CVTSD2SI—Convert with Truncation Scalar Double-Precision FP Value to Signed Integer .....	3-224
CVTSS2SI—Convert with Truncation Scalar Single-Precision FP Value to Dword Integer .....	3-226
CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword .....	3-228
DAA—Decimal Adjust AL after Addition .....	3-229
DAS—Decimal Adjust AL after Subtraction .....	3-231
DEC—Decrement by 1 .....	3-233
DIV—Unsigned Divide .....	3-235
DIVPD—Divide Packed Double-Precision Floating-Point Values .....	3-238
DIVPS—Divide Packed Single-Precision Floating-Point Values .....	3-240
DIVSD—Divide Scalar Double-Precision Floating-Point Values .....	3-242
DIVSS—Divide Scalar Single-Precision Floating-Point Values .....	3-243
DPPD — Dot Product of Packed Double Precision Floating-Point Values .....	3-244
DPPS — Dot Product of Packed Single Precision Floating-Point Values .....	3-246
EMMS—Empty MMX Technology State .....	3-249
ENTER—Make Stack Frame for Procedure Parameters .....	3-251
EXTRACTPS — Extract Packed Single Precision Floating-Point Value .....	3-254
F2XM1—Compute 2 <sup>x</sup> -1 .....	3-256
FABS—Absolute Value .....	3-258
FADD/FADDP/FIADD—Add .....	3-260
FBLD—Load Binary Coded Decimal .....	3-263
FBSTP—Store BCD Integer and Pop .....	3-265
FCHS—Change Sign .....	3-267
FCLEX/FNCLEX—Clear Exceptions .....	3-269
FCMOVcc—Floating-Point Conditional Move .....	3-271
FCOM/FCOMP/FCOMPP—Compare Floating Point Values .....	3-273
FCOMI/FCOMIP/ FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS .....	3-276
FCOS—Cosine .....	3-279
FDECSTP—Decrement Stack-Top Pointer .....	3-281
FDIV/FDIVP/FIDIV—Divide .....	3-282
FDIVR/FDIVRP/FIDIVR—Reverse Divide .....	3-285
FFREE—Free Floating-Point Register .....	3-288
FICOM/FICOMP—Compare Integer .....	3-289
FILD—Load Integer .....	3-291
FINCSTP—Increment Stack-Top Pointer .....	3-293
FINIT/FNINIT—Initialize Floating-Point Unit .....	3-294
FIST/FISTP—Store Integer .....	3-296
FISTTP—Store Integer with Truncation .....	3-299
FLD—Load Floating Point Value .....	3-301
FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant .....	3-303
FLDCW—Load x87 FPU Control Word .....	3-305
FLDENV—Load x87 FPU Environment .....	3-307
FMUL/FMULP/FIMUL—Multiply .....	3-309
FNOP—No Operation .....	3-312
FPATAN—Partial Arctangent .....	3-313
FPREM—Partial Remainder .....	3-315
FPREM1—Partial Remainder .....	3-317
FPTAN—Partial Tangent .....	3-319
FRNDINT—Round to Integer .....	3-321
FRSTOR—Restore x87 FPU State .....	3-322
FSAVE/FNSAVE—Store x87 FPU State .....	3-324
FSCALE—Scale .....	3-327
FSIN—Sine .....	3-329
FSINCOS—Sine and Cosine .....	3-331

	PAGE
FSQRT—Square Root .....	3-333
FST/FSTP—Store Floating Point Value .....	3-335
FSTCW/FNSTCW—Store x87 FPU Control Word .....	3-337
FSTENV/FNSTENV—Store x87 FPU Environment .....	3-339
FSTS/W/FNSTSW—Store x87 FPU Status Word .....	3-341
FSUB/FSUBP/FISUB—Subtract .....	3-343
FSUBR/FSUBRP/FISUBR—Reverse Subtract .....	3-346
FTST—TEST .....	3-349
FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values .....	3-351
FXAM—Examine ModR/M .....	3-353
FXCH—Exchange Register Contents .....	3-355
FXRSTOR—Restore x87 FPU, MMX , XMM, and MXCSR State .....	3-357
FXSAVE—Save x87 FPU, MMX Technology, and SSE State .....	3-360
FXTRACT—Extract Exponent and Significand .....	3-368
FYL2X—Compute $y * \log_2 x$ .....	3-370
FYL2XP1—Compute $y * \log_2(x + 1)$ .....	3-372
HADDPD—Packed Double-FP Horizontal Add .....	3-374
HADDPD—Packed Single-FP Horizontal Add .....	3-377
HLT—Halt .....	3-380
HSUBPD—Packed Double-FP Horizontal Subtract .....	3-381
HSUBPS—Packed Single-FP Horizontal Subtract .....	3-384
IDIV—Signed Divide .....	3-387
IMUL—Signed Multiply .....	3-390
IN—Input from Port .....	3-394
INC—Increment by 1 .....	3-396
INS/INSB/INSW/INSD—Input from Port to String .....	3-398
INSERTPS — Insert Packed Single Precision Floating-Point Value .....	3-401
INT n/INTO/INT 3—Call to Interrupt Procedure .....	3-404
INVD—Invalidate Internal Caches .....	3-416
INVLPG—Invalidate TLB Entry .....	3-418
INVPcid—Invalidate Process-Context Identifier .....	3-420
IRET/IRETd—Interrupt Return .....	3-423
Jcc—Jump if Condition Is Met .....	3-431
JMP—Jump .....	3-436
LAHF—Load Status Flags into AH Register .....	3-444
LAR—Load Access Rights Byte .....	3-445
LDDQU—Load Unaligned Integer 128 Bits .....	3-448
LDMXCSR—Load MXCSR Register .....	3-450
LDS/LES/LFS/LGS/LSS—Load Far Pointer .....	3-451
LEA—Load Effective Address .....	3-455
LEAVE—High Level Procedure Exit .....	3-458
LFENCE—Load Fence .....	3-460
LGDT/LIDT—Load Global/Interrupt Descriptor Table Register .....	3-461
LLDT—Load Local Descriptor Table Register .....	3-464
LMSW—Load Machine Status Word .....	3-466
LOCK—Assert LOCK# Signal Prefix .....	3-468
LODS/LODSB/LODSW/LODSD/LODSQ—Load String .....	3-470
LOOP/LOOPcc—Loop According to ECX Counter .....	3-473
LSL—Load Segment Limit .....	3-475
LTR—Load Task Register .....	3-478
LZCNT— Count the Number of Leading Zero Bits .....	3-480
MASKMOVDQU—Store Selected Bytes of Double Quadword .....	3-482
MASKMOVQ—Store Selected Bytes of Quadword .....	3-484
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values .....	3-486
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values .....	3-488
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value .....	3-490
MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value .....	3-492
MFENCE—Memory Fence .....	3-494
MINPD—Return Minimum Packed Double-Precision Floating-Point Values .....	3-495

## CONTENTS

	PAGE
MINPS—Return Minimum Packed Single-Precision Floating-Point Values .....	3-497
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value .....	3-499
MINSS—Return Minimum Scalar Single-Precision Floating-Point Value .....	3-501
MONITOR—Set Up Monitor Address.....	3-503
MOV—Move .....	3-505
MOV—Move to/from Control Registers.....	3-510
MOV—Move to/from Debug Registers.....	3-513
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values.....	3-515
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values.....	3-517
MOVBE—Move Data After Swapping Bytes.....	3-519
MOVD/MOVQ—Move Doubleword/Move Quadword.....	3-521
MOVDDUP—Move One Double-FP and Duplicate .....	3-524
MOVDQA—Move Aligned Double Quadword .....	3-526
MOVDQU—Move Unaligned Double Quadword.....	3-528
MOVdq2Q—Move Quadword from XMM to MMX Technology Register .....	3-530
MOVHLPS— Move Packed Single-Precision Floating-Point Values High to Low.....	3-531
MOVHPD—Move High Packed Double-Precision Floating-Point Value.....	3-533
MOVHPS—Move High Packed Single-Precision Floating-Point Values.....	3-535
MOVlHPS—Move Packed Single-Precision Floating-Point Values Low to High .....	3-537
MOVLPD—Move Low Packed Double-Precision Floating-Point Value .....	3-539
MOVLPs—Move Low Packed Single-Precision Floating-Point Values .....	3-541
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask.....	3-543
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask.....	3-545
MOVNTDQA — Load Double Quadword Non-Temporal Aligned Hint.....	3-547
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint.....	3-549
MOVNTI—Store Doubleword Using Non-Temporal Hint.....	3-551
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint .....	3-553
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint .....	3-555
MOVNTQ—Store of Quadword Using Non-Temporal Hint.....	3-557
MOVQ—Move Quadword .....	3-558
MOVQ2DQ—Move Quadword from MMX Technology to XMM Register .....	3-560
MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data from String to String .....	3-561
MOVSD—Move Scalar Double-Precision Floating-Point Value .....	3-565
MOVSHDUP—Move Packed Single-FP High and Duplicate .....	3-567
MOVSLDUP—Move Packed Single-FP Low and Duplicate .....	3-569
MOVSS—Move Scalar Single-Precision Floating-Point Values .....	3-571
MOVsx/MOVsxd—Move with Sign-Extension .....	3-573
MOVupd—Move Unaligned Packed Double-Precision Floating-Point Values .....	3-575
MOVups—Move Unaligned Packed Single-Precision Floating-Point Values .....	3-577
MOVzx—Move with Zero-Extend .....	3-579
MPSADBw — Compute Multiple Packed Sums of Absolute Difference .....	3-581
MUL—Unsigned Multiply .....	3-589
MULPD—Multiply Packed Double-Precision Floating-Point Values .....	3-591
MULPS—Multiply Packed Single-Precision Floating-Point Values .....	3-593
MULSD—Multiply Scalar Double-Precision Floating-Point Values .....	3-595
MULSS—Multiply Scalar Single-Precision Floating-Point Values .....	3-596
MULX — Unsigned Multiply Without Affecting Flags .....	3-597
MWAIT—Monitor Wait .....	3-599

## CHAPTER 4

### INSTRUCTION SET REFERENCE, N-Z

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 .....	4-3
4.1.5   Output Selection .....	4-3
4.1.6   Valid/Invalid Override of Comparisons .....	4-3
4.1.7   Summary of Im8 Control byte .....	4-4

	PAGE
4.1.8      Diagram Comparison and Aggregation Process .....	4-5
4.2      INSTRUCTIONS (N-Z) .....	4-5
NEG—Two's Complement Negation .....	4-6
NOP—No Operation .....	4-8
NOT—One's Complement Negation .....	4-9
OR—Logical Inclusive OR .....	4-11
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values .....	4-13
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values .....	4-15
OUT—Output to Port .....	4-17
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port .....	4-19
PABSB/PABSW/PABSD — Packed Absolute Value .....	4-23
PACKSSWB/PACKSSDW—Pack with Signed Saturation .....	4-27
PACKUSDW — Pack with Unsigned Saturation .....	4-32
PACKUSWB—Pack with Unsigned Saturation .....	4-35
PADDB/PADDW/PADDD—Add Packed Integers .....	4-38
PADDQ—Add Packed Quadword Integers .....	4-42
PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation .....	4-44
PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation .....	4-47
PALIGNR — Packed Align Right .....	4-50
PAND—Logical AND .....	4-53
PANDN—Logical AND NOT .....	4-55
PAUSE—Spin Loop Hint .....	4-57
PAVGB/PAVGW—Average Packed Integers .....	4-58
PBLENDVB — Variable Blend Packed Bytes .....	4-61
PBLENDW — Blend Packed Words .....	4-65
PCLMULQDQ - Carry-Less Multiplication Quadword .....	4-68
PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal .....	4-71
PCMPEQQ — Compare Packed Qword Data for Equal .....	4-75
PCMPESTRI — Packed Compare Explicit Length Strings, Return Index .....	4-77
PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask .....	4-79
PCMPGTB/PCMPGTw/PCMPGTD—Compare Packed Signed Integers for Greater Than .....	4-81
PCMPGTQ — Compare Packed Data for Greater Than .....	4-85
PCMPISTRI — Packed Compare Implicit Length Strings, Return Index .....	4-87
PCMPISTRM — Packed Compare Implicit Length Strings, Return Mask .....	4-89
PDEP — Parallel Bits Deposit .....	4-91
PEXT — Parallel Bits Extract .....	4-93
PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword .....	4-95
PEXTRW—Extract Word .....	4-98
PHADDW/PHADD — Packed Horizontal Add .....	4-101
PHADDSW — Packed Horizontal Add and Saturate .....	4-105
PHMINPOSUW — Packed Horizontal Word Minimum .....	4-107
PHSUBW/PHSUBD — Packed Horizontal Subtract .....	4-109
PHSUBSW — Packed Horizontal Subtract and Saturate .....	4-112
PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword .....	4-114
PINSRW—Insert Word .....	4-116
PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes .....	4-118
PMADDWD—Multiply and Add Packed Integers .....	4-120
PMAXSB — Maximum of Packed Signed Byte Integers .....	4-123
PMAXSD — Maximum of Packed Signed Dword Integers .....	4-126
PMAXSW—Maximum of Packed Signed Word Integers .....	4-128
PMAXUB—Maximum of Packed Unsigned Byte Integers .....	4-131
PMAXUD — Maximum of Packed Unsigned Dword Integers .....	4-134
PMAXUW — Maximum of Packed Word Integers .....	4-136
PMINSB — Minimum of Packed Signed Byte Integers .....	4-138
PMINSD — Minimum of Packed Dword Integers .....	4-141
PMINSW—Minimum of Packed Signed Word Integers .....	4-143
PMINUB—Minimum of Packed Unsigned Byte Integers .....	4-146
PMINUD — Minimum of Packed Dword Integers .....	4-149
PMINUW — Minimum of Packed Word Integers .....	4-151

## CONTENTS

	PAGE
PMOVMSKB—Move Byte Mask.....	4-153
PMOVSX — Packed Move with Sign Extend.....	4-155
PMOVZX — Packed Move with Zero Extend.....	4-159
PMULDQ — Multiply Packed Signed Dword Integers.....	4-163
PMULHRSW — Packed Multiply High with Round and Scale .....	4-165
PMULHUW—Multiply Packed Unsigned Integers and Store High Result.....	4-168
PMULHW—Multiply Packed Signed Integers and Store High Result.....	4-172
PMULLD — Multiply Packed Signed Dword Integers and Store Low Result.....	4-175
PMULLW—Multiply Packed Signed Integers and Store Low Result .....	4-177
PMULUDQ—Multiply Packed Unsigned Doubleword Integers.....	4-180
POP—Pop a Value from the Stack.....	4-182
POPA/POPAD—Pop All General-Purpose Registers.....	4-187
POPCNT — Return the Count of Number of Bits Set to 1.....	4-189
POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register .....	4-191
POR—Bitwise Logical OR .....	4-194
PREFETCHh—Prefetch Data Into Caches .....	4-196
PREFETCHW—Prefetch Data into Caches in Anticipation of a Write .....	4-198
PSADBW—Compute Sum of Absolute Differences .....	4-200
PSHUFB — Packed Shuffle Bytes .....	4-203
PSHUFD—Shuffle Packed Doublewords .....	4-206
PSHUFHW—Shuffle Packed High Words .....	4-208
PSHUFLW—Shuffle Packed Low Words .....	4-210
PSHUFW—Shuffle Packed Words .....	4-212
PSIGNB/PSIGNW/PSIGND — Packed SIGN .....	4-213
PSLLDQ—Shift Double Quadword Left Logical .....	4-217
PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical .....	4-219
PSRAW/PSRAD—Shift Packed Data Right Arithmetic .....	4-225
PSRLDQ—Shift Double Quadword Right Logical .....	4-230
PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical .....	4-232
PSUBB/PSUBW/PSUBD—Subtract Packed Integers .....	4-238
PSUBQ—Subtract Packed Quadword Integers .....	4-243
PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation .....	4-245
PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation .....	4-248
PTEST- Logical Compare .....	4-251
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data .....	4-253
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data .....	4-260
PUSH—Push Word, Doubleword or Quadword Onto the Stack .....	4-267
PUSHA/PUSHAD—Push All General-Purpose Registers .....	4-270
PUSHF/PUSHFD—Push EFLAGS Register onto the Stack .....	4-272
PXOR—Logical Exclusive OR .....	4-274
RCL/RCR/ROL/ROR—Rotate .....	4-276
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values .....	4-281
RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values .....	4-283
RDFSBASE/RDGSBASE—Read FS/GS Segment Base .....	4-285
RDMSR—Read from Model Specific Register .....	4-287
RDPMC—Read Performance-Monitoring Counters .....	4-289
RDRAND—Read Random Number .....	4-293
RDTSC—Read Time-Stamp Counter .....	4-295
RDTSCP—Read Time-Stamp Counter and Processor ID .....	4-297
REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix .....	4-299
RET—Return from Procedure .....	4-302
RORX — Rotate Right Logical Without Affecting Flags .....	4-312
ROUNDPD — Round Packed Double Precision Floating-Point Values .....	4-313
ROUNDPS — Round Packed Single Precision Floating-Point Values .....	4-316
ROUNDSD — Round Scalar Double Precision Floating-Point Values .....	4-319
ROUNDSS — Round Scalar Single Precision Floating-Point Values .....	4-321
RSM—Resume from System Management Mode .....	4-323
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values .....	4-325
RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value .....	4-327

	PAGE
SAHF—Store AH into Flags.....	4-329
SAL/SAR/SHL/SHR—Shift .....	4-331
SARX/SHLX/SHRX – Shift Without Affecting Flags.....	4-336
SBB—Integer Subtraction with Borrow .....	4-338
SCAS/SCASB/SCASW/SCASD—Scan String .....	4-341
SETcc—Set Byte on Condition.....	4-345
SFENCE—Store Fence.....	4-348
SGDT—Store Global Descriptor Table Register.....	4-349
SHLD—Double Precision Shift Left.....	4-351
SHRD—Double Precision Shift Right .....	4-354
SHUFPD—Shuffle Packed Double-Precision Floating-Point Values .....	4-357
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values .....	4-360
SIDT—Store Interrupt Descriptor Table Register .....	4-363
SLDT—Store Local Descriptor Table Register.....	4-365
SMSW—Store Machine Status Word .....	4-367
SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values .....	4-369
SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values .....	4-371
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value.....	4-373
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value.....	4-374
STC—Set Carry Flag .....	4-375
STD—Set Direction Flag.....	4-376
STI—Set Interrupt Flag.....	4-377
STMXCSR—Store MXCSR Register State .....	4-379
STOS/STOSB/STOSW/STOSD/STOSQ—Store String.....	4-380
STR—Store Task Register.....	4-384
SUB—Subtract .....	4-386
SUBPD—Subtract Packed Double-Precision Floating-Point Values .....	4-388
SUBPS—Subtract Packed Single-Precision Floating-Point Values .....	4-390
SUBSD—Subtract Scalar Double-Precision Floating-Point Values .....	4-392
SUBSS—Subtract Scalar Single-Precision Floating-Point Values.....	4-393
SWAPGS—Swap GS Base Register .....	4-394
SYSCALL—Fast System Call .....	4-396
SYSENTER—Fast System Call.....	4-398
SYSEXIT—Fast Return from Fast System Call .....	4-401
SYSRET—Return From Fast System Call .....	4-404
TEST—Logical Compare .....	4-407
TZCNT — Count the Number of Trailing Zero Bits .....	4-409
UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS .....	4-411
UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS .....	4-413
UD2—Undefined Instruction.....	4-415
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values.....	4-416
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values.....	4-418
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values .....	4-420
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values .....	4-422
VBROADCAST—Broadcast Floating-Point Data .....	4-424
VCVTPH2PS—Convert 16-bit FP Values to Single-Precision FP Values.....	4-428
VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value .....	4-430
VERR/VERW—Verify a Segment for Reading or Writing.....	4-433
VEXTRACTF128 — Extract Packed Floating-Point Values.....	4-435
VEXTRACTI128 — Extract packed Integer Values .....	4-436
VFMADD132PD/VFMADD213PD/VFMADD231PD — Fused Multiply-Add of Packed Double-Precision Floating-Point Values .....	4-437
VFMADD132PS/VFMADD213PS/VFMADD231PS — Fused Multiply-Add of Packed Single-Precision Floating-Point Values .....	4-440
VFMADD132SD/VFMADD213SD/VFMADD231SD — Fused Multiply-Add of Scalar Double-Precision Floating-Point Values .....	4-443
VFMADD132SS/VFMADD213SS/VFMADD231SS — Fused Multiply-Add of Scalar Single-Precision Floating-Point Values .....	4-445
VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD — Fused Multiply-Alternating Add/Subtract of Packed	

## CONTENTS

	PAGE
Double-Precision Floating-Point Values .....	4-447
VFMADDSD132PS/VFMADDSD213PS/VFMADDSD231PS — Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values .....	4-450
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD — Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values .....	4-453
VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS — Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values .....	4-456
VFMSUB132PD/VFMSUB213PD/VFMSUB231PD — Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values .....	4-459
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS — Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values .....	4-462
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD — Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values .....	4-465
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS — Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values .....	4-467
VFNMADD132PD/VFNMADD213PD/VFNMADD231PD — Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values .....	4-469
VFNMADD132PS/VFNMADD213PS/VFNMADD231PS — Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values .....	4-472
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD — Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values .....	4-475
VFNMADD132SS/VFNMADD213SS/VFNMADD231SS — Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values .....	4-477
VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD — Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values .....	4-479
VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS — Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values .....	4-482
VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD — Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values .....	4-485
VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS — Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values .....	4-487
VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices .....	4-489
VGATHERDPS/VGATHEROPS — Gather Packed SP FP values Using Signed Dword/Qword Indices .....	4-493
VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices .....	4-497
VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices .....	4-501
VINSERTF128 — Insert Packed Floating-Point Values .....	4-505
VINSERTI128 — Insert Packed Integer Values .....	4-506
VMASKMOV—Conditional SIMD Packed Loads and Stores .....	4-507
VPBLENDD — Blend Packed Dwors .....	4-510
VPBROADCAST—Broadcast Integer Data .....	4-512
VPERMD — Full Doublewords Element Permutation .....	4-516
VPERMPD — Permute Double-Precision Floating-Point Elements .....	4-517
VPERMPS — Permute Single-Precision Floating-Point Elements .....	4-518
VPERMQ — Qwords Element Permutation .....	4-519
VPERM2I128 — Permute Integer Values .....	4-520
VPERMILPD — Permute Double-Precision Floating-Point Values .....	4-522
VPERMILPS — Permute Single-Precision Floating-Point Values .....	4-525
VPERM2F128 — Permute Floating-Point Values .....	4-528
VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores .....	4-530
VPSLLVD/VPSLLVQ — Variable Bit Shift Left Logical .....	4-533
VPSRAVD — Variable Bit Shift Right Arithmetic .....	4-535
VPSRLVD/VPSRLVQ — Variable Bit Shift Right Logical .....	4-537
VTESTPD/VTESTPS—Packed Bit Test .....	4-539
VZEROALL—Zero All YMM Registers .....	4-542
VZEROUPPER—Zero Upper Bits of YMM Registers .....	4-544
WAIT/FWAIT—Wait .....	4-546
WBINVD—Write Back and Invalidate Cache .....	4-547
WRFSBASE/WRGSBASE—Write FS/GS Segment Base .....	4-549
WRMSR—Write to Model Specific Register .....	4-551

	PAGE
XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints .....	4-553
XABORT — Transactional Abort .....	4-556
XADD—Exchange and Add .....	4-558
XBEGIN — Transactional Begin .....	4-560
XCHG—Exchange Register/Memory with Register .....	4-563
XEND — Transactional End .....	4-565
XGETBV—Get Value of Extended Control Register .....	4-567
XLAT/XLATB—Table Look-up Translation .....	4-569
XOR—Logical Exclusive OR .....	4-571
XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values .....	4-573
XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values .....	4-575
XRSTOR—Restore Processor Extended States .....	4-577
XRSTORS—Restore Processor Extended States Supervisor .....	4-581
XSAVE—Save Processor Extended States .....	4-585
XSAVEC—Save Processor Extended States with Compaction .....	4-588
XSAVEOPT—Save Processor Extended States Optimized .....	4-591
XSAVES—Save Processor Extended States Supervisor .....	4-594
XSETBV—Set Extended Control Register .....	4-597
XTEST — Test If In Transactional Execution .....	4-599

**CHAPTER 5****SAFER MODE EXTENSIONS REFERENCE**

5.1 OVERVIEW .....	5-1
5.2 SMX FUNCTIONALITY .....	5-1
5.2.1 Detecting and Enabling SMX .....	5-1
5.2.2 SMX Instruction Summary .....	5-2
5.2.2.1 GETSEC[CAPABILITIES] .....	5-2
5.2.2.2 GETSEC[ENTERACCS] .....	5-3
5.2.2.3 GETSEC[EXITAC] .....	5-3
5.2.2.4 GETSEC[SETER] .....	5-3
5.2.2.5 GETSEC[SEXIT] .....	5-4
5.2.2.6 GETSEC[PARAMETERS] .....	5-4
5.2.2.7 GETSEC[SMCTRL] .....	5-4
5.2.2.8 GETSEC[WAKEUP] .....	5-4
5.2.3 Measured Environment and SMX .....	5-4
5.3 GETSEC LEAF FUNCTIONS .....	5-5
GETSEC[CAPABILITIES] - Report the SMX Capabilities .....	5-7
GETSEC[ENTERACCS] - Execute Authenticated Chipset Code .....	5-9
GETSEC[EXITAC]—Exit Authenticated Code Execution Mode .....	5-17
GETSEC[SETER]—Enter a Measured Environment .....	5-20
GETSEC[SEXIT]—Exit Measured Environment .....	5-29
GETSEC[PARAMETERS]—Report the SMX Parameters .....	5-32
GETSEC[SMCTRL]—SMX Mode Control .....	5-36
GETSEC[WAKEUP]—Wake up sleeping processors in measured environment .....	5-39

**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

## CONTENTS

	PAGE
A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES .....	A-18
A.4.1   Opcodes Look-up Examples Using Opcode Extensions .....	A-18
A.4.2   Opcode Extension Tables .....	A-18
A.5 ESCAPE OPCODE INSTRUCTIONS .....	A-21
A.5.1   Opcodes 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-2
B.1.4.2   Reg Field (reg) for 64-Bit Mode .....	B-3
B.1.4.3   Encoding of Operand Size (w) Bit .....	B-4
B.1.4.4   Sign-Extend (s) Bit .....	B-4
B.1.4.5   Segment Register (sreg) Field .....	B-4
B.1.4.6   Special-Purpose Register (eee) Field .....	B-5
B.1.4.7   Condition Test (tttn) Field .....	B-5
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-48
B.9.1   Granularity Field (gg) .....	B-48
B.10 SSE3 FORMATS AND ENCODINGS TABLE .....	B-59
B.11 SSSE3 FORMATS AND ENCODING TABLE .....	B-60
B.12 AESNI AND PCLMULQDQ INSTRUCTION FORMATS AND ENCODINGS .....	B-63
B.13 SPECIAL ENCODINGS FOR 64-BIT MODE .....	B-64
B.14 SSE4.1 FORMATS AND ENCODING TABLE .....	B-66
B.15 SSE4.2 FORMATS AND ENCODING TABLE .....	B-71
B.16 AVX FORMATS AND ENCODING TABLE .....	B-73
B.17 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS .....	B-113
B.18 VMX INSTRUCTIONS .....	B-117
B.19 SMX INSTRUCTIONS .....	B-118

## APPENDIX C

### INTEL® C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

C.1 SIMPLE INTRINSICS .....	C-2
C.2 COMPOSITE INTRINSICS .....	C-14

## FIGURES

Figure 1-1.	Bit and Byte Order .....	1-4
Figure 1-2.	Syntax for CPUID, CR, and MSR Data Presentation.....	1-6
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 .....	2-7
Figure 2-4.	Memory Addressing Without an SIB Byte; REX.X Not Used.....	2-8
Figure 2-5.	Register-Register Addressing (No Memory Operand); REX.X Not Used.....	2-8
Figure 2-6.	Memory Addressing With a SIB Byte.....	2-9
Figure 2-7.	Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used.....	2-9
Figure 2-8.	Instruction Encoding Format with VEX Prefix .....	2-12
Figure 2-9.	VEX bitfields .....	2-14
Figure 3-1.	Bit Offset for BIT[RAX, 21] .....	3-10
Figure 3-2.	Memory Bit Indexing.....	3-11
Figure 3-3.	ADDSUBPD—Packed Double-FP Add/Subtract.....	3-36
Figure 3-4.	ADDSUBPS—Packed Single-FP Add/Subtract .....	3-38
Figure 3-5.	Version Information Returned by CPUID in EAX .....	3-168
Figure 3-6.	Feature Information Returned in the ECX Register .....	3-170
Figure 3-7.	Feature Information Returned in the EDX Register.....	3-172
Figure 3-8.	Determination of Support for the Processor Brand String.....	3-180
Figure 3-9.	Algorithm for Extracting Maximum Processor Frequency .....	3-182
Figure 3-10.	CVTDQ2PD (VEX.256 encoded version).....	3-191
Figure 3-11.	VCVTTPD2DQ (VEX.256 encoded version) .....	3-196
Figure 3-12.	VCVTTPD2PS (VEX.256 encoded version).....	3-198
Figure 3-13.	CVTPS2PD (VEX.256 encoded version).....	3-204
Figure 3-14.	VCVTTPD2DQ (VEX.256 encoded version) .....	3-219
Figure 3-15.	HADDPD—Packed Double-FP Horizontal Add .....	3-374
Figure 3-16.	VHADDPD operation.....	3-375
Figure 3-17.	HADDPS—Packed Single-FP Horizontal Add .....	3-378
Figure 3-18.	VHADDPS operation .....	3-378
Figure 3-19.	HSUBPD—Packed Double-FP Horizontal Subtract .....	3-381
Figure 3-20.	VHSUBPD operation .....	3-382
Figure 3-21.	HSUBPS—Packed Single-FP Horizontal Subtract .....	3-385
Figure 3-22.	VHSUBPS operation .....	3-385
Figure 3-23.	INVPcid Descriptor .....	3-420
Figure 3-24.	MOVDDUP—Move One Double-FP and Duplicate .....	3-524
Figure 3-25.	MOVSHDUP—Move Packed Single-FP High and Duplicate .....	3-567
Figure 3-26.	MOVSLDUP—Move Packed Single-FP Low and Duplicate .....	3-569
Figure 3-27.	VMPSADBW Operation.....	3-582
Figure 4-1.	Operation of PCMPSTRx and PCMPESTRx.....	4-5
Figure 4-2.	Operation of the PACKSSDW Instruction Using 64-bit Operands.....	4-28
Figure 4-3.	256-bit VPALIGN Instruction Operation .....	4-51
Figure 4-4.	PDEP Example .....	4-91
Figure 4-5.	PEXT Example .....	4-93
Figure 4-6.	256-bit VPHADD Instruction Operation .....	4-102
Figure 4-7.	PMADDWD Execution Model Using 64-bit Operands .....	4-121
Figure 4-8.	PMULHUW and PMULHW Instruction Operation Using 64-bit Operands .....	4-169
Figure 4-9.	PMULLU Instruction Operation Using 64-bit Operands .....	4-178
Figure 4-10.	PSADBW Instruction Operation Using 64-bit Operands.....	4-201
Figure 4-11.	PSHUFW with 64-Bit Operands .....	4-205
Figure 4-12.	256-bit VPSHUFD Instruction Operation .....	4-206
Figure 4-13.	PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand .....	4-220
Figure 4-14.	PSRAW and PSRAD Instruction Operation Using a 64-bit Operand .....	4-226
Figure 4-15.	PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand .....	4-233
Figure 4-16.	PUNPCKHBW Instruction Operation Using 64-bit Operands .....	4-254
Figure 4-17.	256-bit VPUNPCKHDQ Instruction Operation.....	4-254
Figure 4-18.	PUNPCKLBW Instruction Operation Using 64-bit Operands .....	4-261
Figure 4-19.	256-bit VPUNPCKLDQ Instruction Operation .....	4-261

## CONTENTS

	PAGE
Figure 4-20. Bit Control Fields of Immediate Byte for ROUNDxx Instruction .....	4-314
Figure 4-21. SHUFPD Shuffle Operation .....	4-357
Figure 4-22. SHUFPD Shuffle Operation .....	4-360
Figure 4-23. UNPCKHPD Instruction High Unpack and Interleave Operation .....	4-416
Figure 4-24. UNPCKHPS Instruction High Unpack and Interleave Operation .....	4-418
Figure 4-25. UNPCKLPD Instruction Low Unpack and Interleave Operation .....	4-420
Figure 4-26. UNPCKLPS Instruction Low Unpack and Interleave Operation .....	4-422
Figure 4-27. VBROADCASTSS Operation (VEX.256 encoded version).....	4-425
Figure 4-28. VBROADCASTSS Operation (128-bit version).....	4-425
Figure 4-29. VBROADCASTSD Operation .....	4-425
Figure 4-30. VBROADCASTF128 Operation.....	4-426
Figure 4-31. VCVTPH2PS (128-bit Version). ....	4-428
Figure 4-32. VCVTPS2PH (128-bit Version). ....	4-430
Figure 4-33. VPBROADCASTD Operation (VEX.256 encoded version) .....	4-513
Figure 4-34. VPBROADCASTD Operation (128-bit version) .....	4-513
Figure 4-35. VPBROADCASTQ Operation .....	4-513
Figure 4-36. VBROADCASTI128 Operation .....	4-514
Figure 4-37. VPERM2I128 Operation.....	4-520
Figure 4-38. VPERMILPD operation .....	4-522
Figure 4-39. VPERMILPD Shuffle Control .....	4-523
Figure 4-40. VPERMILPS Operation .....	4-525
Figure 4-41. VPERMILPS Shuffle Control .....	4-526
Figure 4-42. VPERM2F128 Operation .....	4-528
Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3).....	A-18
Figure B-1. General Machine Instruction Format .....	B-1
Figure B-2. Hybrid Notation of VEX-Encoded Key Instruction Bytes.....	B-73

**TABLES**

Table 2-1.	16-Bit Addressing Forms with the ModR/M Byte .....	2-4
Table 2-2.	32-Bit Addressing Forms with the ModR/M Byte .....	2-5
Table 2-3.	32-Bit Addressing Forms with the SIB Byte .....	2-6
Table 2-4.	REX Prefix Fields [BITS: 0100WRXB] .....	2-8
Table 2-6.	Direct Memory Offset Form of MOV .....	2-10
Table 2-5.	Special Cases of REX Encodings .....	2-10
Table 2-7.	RIP-Relative Addressing .....	2-11
Table 2-8.	VEX.vvvv to register name mapping .....	2-16
Table 2-9.	Instructions with a VEX.vvvv destination .....	2-16
Table 2-10.	VEX.m-mmmm interpretation .....	2-17
Table 2-11.	VEX.L interpretation .....	2-17
Table 2-12.	VEX.pp interpretation .....	2-18
Table 2-13.	32-Bit VSIB Addressing Forms of the SIB Byte .....	2-19
Table 2-14.	Exception class description .....	2-21
Table 2-15.	Instructions in each Exception Class .....	2-22
Table 2-16.	#UD Exception and VEX.W=1 Encoding .....	2-23
Table 2-17.	#UD Exception and VEX.L Field Encoding .....	2-24
Table 2-18.	Type 1 Class Exception Conditions .....	2-25
Table 2-19.	Type 2 Class Exception Conditions .....	2-26
Table 2-20.	Type 3 Class Exception Conditions .....	2-27
Table 2-21.	Type 4 Class Exception Conditions .....	2-28
Table 2-22.	Type 5 Class Exception Conditions .....	2-29
Table 2-23.	Type 6 Class Exception Conditions .....	2-30
Table 2-24.	Type 7 Class Exception Conditions .....	2-30
Table 2-25.	Type 8 Class Exception Conditions .....	2-31
Table 2-26.	Type 11 Class Exception Conditions .....	2-32
Table 2-27.	Type 12 Class Exception Conditions .....	2-33
Table 2-28.	VEX-Encoded GPR Instructions .....	2-34
Table 2-29.	Exception Definition (VEX-Encoded GPR Instructions) .....	2-34
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 .....	3-10
Table 3-3.	Intel 64 and IA-32 General Exceptions .....	3-13
Table 3-4.	x87 FPU Floating-Point Exceptions .....	3-15
Table 3-5.	SIMD Floating-Point Exceptions .....	3-15
Table 3-6.	Decision Table for CLI Results .....	3-113
Table 3-7.	Comparison Predicate for CMPPD and CMPPS Instructions .....	3-123
Table 3-8.	Pseudo-Op and CMPPD Implementation .....	3-124
Table 3-9.	Comparison Predicate for VCMPPD and VCMPPS Instructions .....	3-125
Table 3-10.	Pseudo-Op and VCMPPD Implementation .....	3-126
Table 3-11.	Pseudo-Ops and CMPPS .....	3-131
Table 3-12.	Pseudo-Op and VCMPPS Implementation .....	3-131
Table 3-14.	Pseudo-Op and VCMPSD Implementation .....	3-141
Table 3-13.	Pseudo-Ops and CMPSD .....	3-141
Table 3-16.	Pseudo-Op and VCMPSS Implementation .....	3-145
Table 3-15.	Pseudo-Ops and CMPSS .....	3-145
Table 3-17.	Information Returned by CPUID Instruction .....	3-158
Table 3-18.	Highest CPUID Source Operand for Intel 64 and IA-32 Processors .....	3-166
Table 3-19.	Processor Type Field .....	3-169
Table 3-20.	Feature Information Returned in the ECX Register .....	3-170
Table 3-21.	More on Feature Information Returned in the EDX Register .....	3-173
Table 3-22.	Encoding of CPUID Leaf 2 Descriptors .....	3-175
Table 3-23.	Processor Brand String Returned with Pentium 4 Processor .....	3-180
Table 3-24.	Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings .....	3-182
Table 3-25.	DIV Action .....	3-235
Table 3-26.	Results Obtained from F2XM1 .....	3-256
Table 3-27.	Results Obtained from FABS .....	3-258
Table 3-28.	FADD/FADDP/FIADD Results .....	3-261

## CONTENTS

	PAGE
Table 3-29. FBSTP Results.....	3-265
Table 3-30. FCHS Results.....	3-267
Table 3-31. FCOM/FCOMP/FCOMPP Results.....	3-273
Table 3-32. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results .....	3-276
Table 3-33. FCOS Results.....	3-279
Table 3-34. FDIV/FDIVP/FIDIV Results.....	3-283
Table 3-35. FDIVR/FDIVRP/FIDIVR Results.....	3-286
Table 3-36. FICOM/FICOMP Results.....	3-289
Table 3-37. FIST/FISTP Results .....	3-296
Table 3-38. FISTTP Results .....	3-299
Table 3-39. FMUL/FMULP/FIMUL Results.....	3-310
Table 3-40. FPATAN Results.....	3-313
Table 3-41. FPREM Results .....	3-315
Table 3-42. FPREM1 Results.....	3-317
Table 3-43. FPTAN Results .....	3-319
Table 3-44. FSACLE Results.....	3-327
Table 3-45. FSIN Results.....	3-329
Table 3-46. FSINCOS Results.....	3-331
Table 3-47. FSQRT Results .....	3-333
Table 3-48. FSUB/FSUBP/FISUB Results.....	3-344
Table 3-49. FSUBR/FSUBRP/FISUBR Results.....	3-347
Table 3-50. FTST Results.....	3-349
Table 3-51. FUCOM/FUCOMP/FUCOMPP Results.....	3-351
Table 3-52. FXAM Results .....	3-353
Table 3-53. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region.....	3-360
Table 3-54. Field Definitions .....	3-361
Table 3-55. Recreating FSAVE Format.....	3-363
Table 3-56. Layout of the 64-bit-mode FXSAVE64 Map (requires REX.W = 1).....	3-363
Table 3-57. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0).....	3-364
Table 3-58. FYL2X Results.....	3-370
Table 3-59. FYL2XP1 Results.....	3-372
Table 3-60. IDIV Results .....	3-387
Table 3-61. Decision Table.....	3-405
Table 3-62. Segment and Gate Types .....	3-446
Table 3-63. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes.....	3-455
Table 3-64. 64-bit Mode LEA Operation with Address and Operand Size Attributes .....	3-455
Table 3-65. Segment and Gate Descriptor Types.....	3-476
Table 3-66. MUL Results.....	3-589
Table 3-67. MWAIT Extension Register (ECX) .....	3-600
Table 3-68. MWAIT Hints Register (EAX).....	3-600
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. Ouput Selection .....	4-3
Table 4-6. Output Selection.....	4-3
Table 4-7. Comparison Result for Each Element Pair BoolRes[i,j] .....	4-4
Table 4-8. Summary of Imm8 Control Byte .....	4-4
Table 4-9. Recommended Multi-Byte Sequence of NOP Instruction .....	4-8
Table 4-10. PCLMULQDQ Quadword Selection of Immediate Byte .....	4-68
Table 4-11. Pseudo-Op and PCLMULQDQ Implementation.....	4-68
Table 4-12. Valid General and Special Purpose Performance Counter Index Range for RDPMC .....	4-289
Table 4-13. Repeat Prefixes .....	4-300
Table 4-14. Rounding Modes and Encoding of Rounding Control (RC) Field .....	4-314
Table 4-15. Decision Table for STI Results .....	4-377
Table 4-16. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions.....	4-431
Table 5-1. Layout of IA32_FEATURE_CONTROL .....	5-2
Table 5-2. GETSEC Leaf Functions .....	5-3
Table 5-3. Getsec Capability Result Encoding (EBX = 0).....	5-7

	PAGE	
Table 5-4.	Register State Initialization after GETSEC[ENTERACCS].....	5-11
Table 5-5.	IA32_MISC_ENABLE MSR Initialization by ENTERACCS and SENTER .....	5-12
Table 5-6.	Register State Initialization after GETSEC[SENTER] and GETSEC[WAKEUP] .....	5-23
Table 5-7.	SMX Reporting Parameters Format.....	5-32
Table 5-8.	TXT Feature Extensions Flags .....	5-33
Table 5-9.	External Memory Types Using Parameter 3.....	5-34
Table 5-10.	Default Parameter Values .....	5-34
Table 5-11.	Supported Actions for GETSEC[SMCTRL(0)] .....	5-36
Table 5-12.	RLP MVMM JOIN Data Structure .....	5-39
Table A-1.	Superscripts Utilized in Opcode Tables.....	A-6
Table A-2.	One-byte Opcode Map: (00H – F7H) * .....	A-8
Table A-3.	Two-byte Opcode Map: 00H – 77H (First Byte is OFH) * .....	A-10
Table A-4.	Three-byte Opcode Map: 00H – F7H (First Two Bytes are OF 38H) * .....	A-14
Table A-5.	Three-byte Opcode Map: 00H – F7H (First two bytes are OF 3AH) * .....	A-16
Table A-6.	Opcode Extensions for One- and Two-byte Opcodes by Group Number *	A-19
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
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-3
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-4
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-11.	Encoding of Operation Direction (d) Bit .....	B-6
Table B-12.	Notes on Instruction Encoding .....	B-6
Table B-10.	Encoding of Conditional Test (tttn) Field .....	B-6
Table B-13.	General Purpose Instruction Formats and Encodings for Non-64-Bit Modes .....	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-47
Table B-25.	Encoding of Granularity of Data Field (gg).....	B-48
Table B-24.	Format and Encoding of SSE Cacheability & Memory Ordering Instructions.....	B-48
Table B-26.	Formats and Encodings of SSE2 Floating-Point Instructions .....	B-49
Table B-27.	Formats and Encodings of SSE2 Integer Instructions .....	B-54
Table B-28.	Format and Encoding of SSE2 Cacheability Instructions .....	B-58

## CONTENTS

	PAGE
Table B-29. Formats and Encodings of SSE3 Floating-Point Instructions .....	B-59
Table B-30. Formats and Encodings for SSE3 Event Management Instructions .....	B-59
Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions.....	B-60
Table B-32. Formats and Encodings for SSSE3 Instructions .....	B-60
Table B-33. Formats and Encodings of AESNI and PCLMULQDQ Instructions .....	B-63
Table B-34. Special Case Instructions Promoted Using REX.W .....	B-64
Table B-35. Encodings of SSE4.1 instructions .....	B-66
Table B-36. Encodings of SSE4.2 instructions .....	B-72
Table B-37. Encodings of AVX instructions.....	B-73
Table B-38. General Floating-Point Instruction Formats.....	B-113
Table B-39. Floating-Point Instruction Formats and Encodings .....	B-113
Table B-40. Encodings for VMX Instructions .....	B-117
Table B-41. Encodings for SMX Instructions.....	B-118
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: Instruction Set Reference* (order numbers 253666, 253667 and 326018) are 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: System Programming Guide* (order numbers 253668, 253669 and 326019).

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*, describe 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*, describe 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.

### 1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series

## ABOUT THIS MANUAL

- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v3 product family

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® micro-architecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processor family is based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Intel® microarchitecture code name Ivy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families and Intel® Xeon® processor E5-2400/1400 v2 product families are based on the Intel® microarchitecture code name Ivy Bridge-EP and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Intel® microarchitecture code name Haswell and support Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

## 1.2 OVERVIEW OF VOLUME 2A, 2B AND 2C: INSTRUCTION SET REFERENCE

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

**Chapter 1 — About This Manual.** Gives an overview of all seven volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel® manuals and documentation of interest to programmers and hardware designers.

**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-Z.** Continues the description of Intel 64 and IA-32 instructions started in Chapter 3. It provides the balance of the alphabetized list of instructions and starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

**Chapter 5—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. This chapter starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*.

**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.

## 1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

### 1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

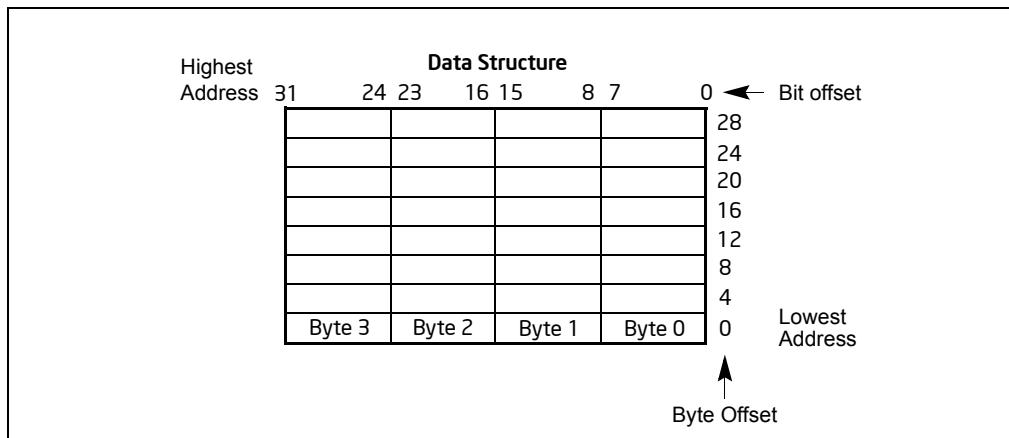


Figure 1-1. Bit and Byte Order

### 1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

#### NOTE

Avoid any software dependence upon the state of reserved bits in IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

### 1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

label: mnemonic argument1, argument2, argument3

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands *argument1*, *argument2*, and *argument3* are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items.

Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

LOADREG: MOV EAX, SUBTOTAL

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

### 1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes in memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

### 1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

### 1.3.7 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

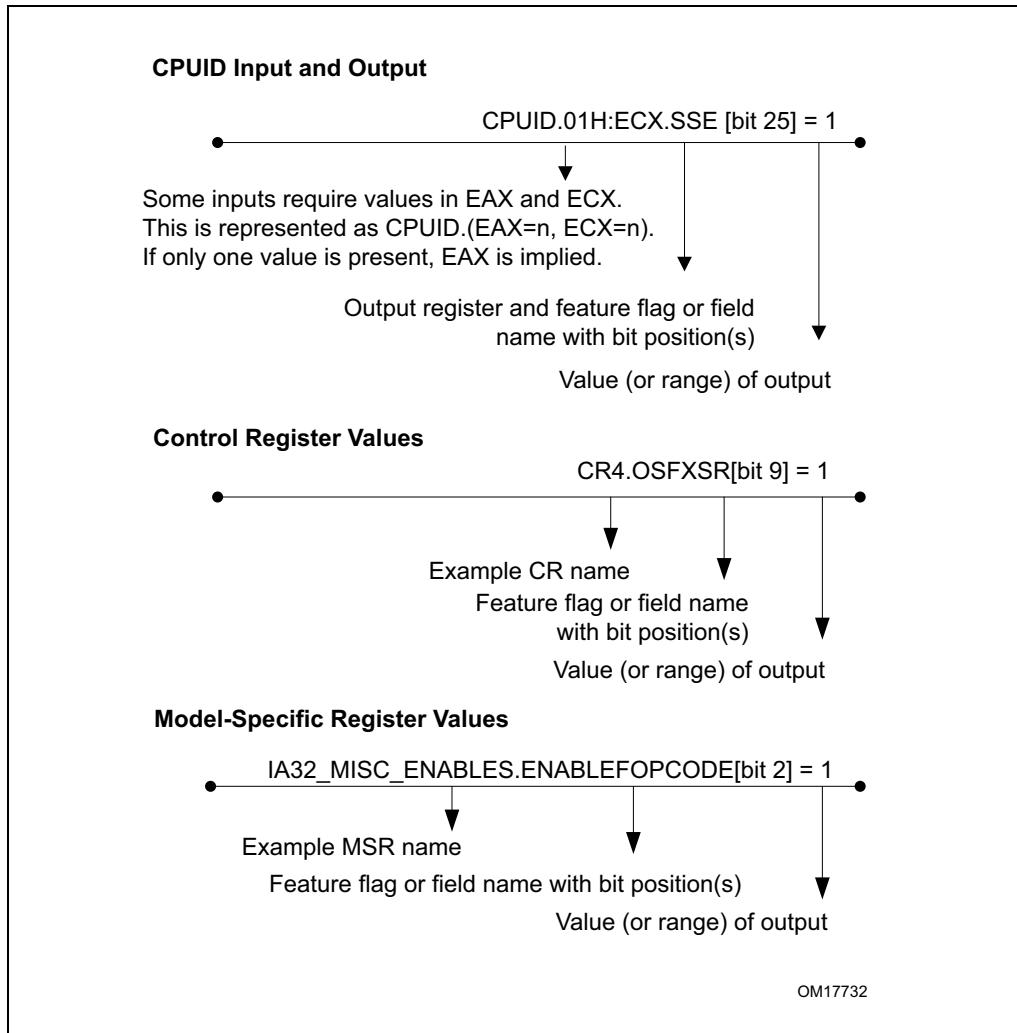


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed on-line at:

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.html>

Some of the documents listed at this web site can be viewed on-line; others can be ordered. The literature available is listed by Intel processor and then by the following literature types: applications notes, data sheets, manuals, papers, and specification updates.

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor

- Intel® C++ Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Fortran Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® VTune™ Performance Analyzer documentation and online help:  
<http://www.intel.com/cd/software/products/asmo-na/eng/index.htm>
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in three or five volumes):  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.html>
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:  
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>
- Intel® Processor Identification with the CPUID Instruction, AP-485:  
<http://www.intel.com/Assets/PDF/appnote/241618.pdf>
- Intel 64 Architecture x2APIC Specification:  
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-architecture-x2apic-specification.html>
- Intel 64 Architecture Processor Topology Enumeration:  
<http://softwarecommunity.intel.com/articles/eng/3887.htm>
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:  
<http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html>
- Intel® SSE4 Programming Reference: <http://edc.intel.com/Link.aspx?id=1630&wapkw=intel® sse4 programming reference>
- Developing Multi-threaded Applications: A Platform Consistent Approach:  
[http://cache-www.intel.com/cd/00/00/05/15/51534\\_developing\\_multithreaded\\_applications.pdf](http://cache-www.intel.com/cd/00/00/05/15/51534_developing_multithreaded_applications.pdf)
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:  
<http://software.intel.com/en-us/articles/ap949-using-spin-loops-on-intel-pentiumr-4-processor-and-intel-xeonr-processor/>
- Performance Monitoring Unit Sharing Guide  
<http://software.intel.com/file/30388>

More relevant links are:

- Software network link:  
<http://softwarecommunity.intel.com/isn/home/>
- Developer centers:  
<http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/index.htm>
- Processor support general link:  
<http://www.intel.com/support/processors/>
- Software products and packages:  
<http://www.intel.com/cd/software/products/asmo-na/eng/index.htm>
- Intel 64 and IA-32 processor manuals (printed or PDF downloads):  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.html>
- Intel® Multi-Core Technology:  
<http://software.intel.com/partner/multicore>
- Intel® Hyper-Threading Technology (Intel® HT Technology):  
<http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>

## ABOUT THIS MANUAL

# 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).

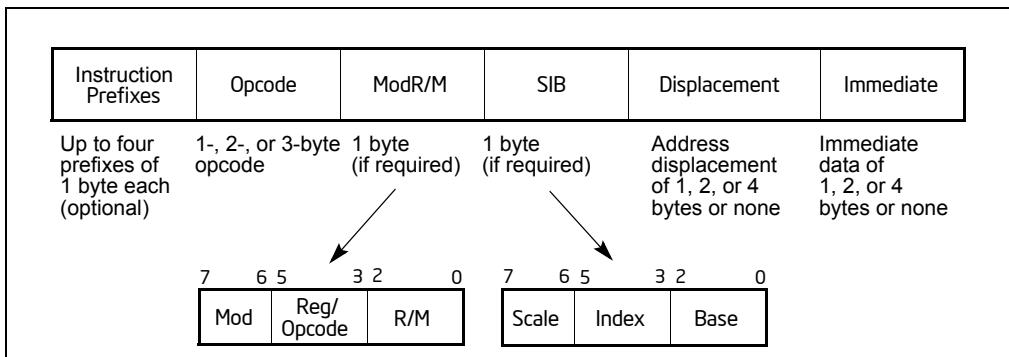


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

#### 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/REPNEZ 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. Repeat prefix applies only to string and input/output instructions. (F3H is also used as a mandatory prefix for some instructions)
- 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)

## INSTRUCTION FORMAT

- 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:
  - 2EH—Branch not taken (used only with Jcc instructions)
  - 3EH—Branch taken (used only with Jcc 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-M,” 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,F3H as a mandatory prefix to express distinct functionality. A mandatory prefix generally should be placed after other optional prefixes (exception to this is discussed in Section 2.2.1, “REX Prefixes”)

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch. Use these prefixes only with conditional branch instructions (Jcc). Other use of branch hint prefixes and/or other undefined opcodes with Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

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. A mandatory prefix generally should be placed after other optional prefixes (exception to this is discussed in Section 2.2.1, “REX Prefixes”)

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, 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:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte, or
- 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 OF 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:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes, or

- 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 is 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 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 MMOV; 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.

## INSTRUCTION FORMAT

If the instruction does not require a second operand, then the Reg/Opc 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.

Mod    11	
RM        000	
/digit (Opcode); REG = 001	
C8H 11001000	

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

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

Effective Address			Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
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 BP1 EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111	
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16 <sup>2</sup> [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 <sup>3</sup> [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 D1 CA CB D2 D3 D4	D0 D9 DA DB DC D5 D6 D7	D8 EQ E2 E3 E4 DD DE DF	E0 E9 EA EB 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**

			AL	CL	DL	BL	AH	CH	DH	BH
			AX	CX	DX	BX	SP	BP	SI	DI
			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	011	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--] <sup>1</sup>		100	04	0C	14	1C	24	2C	34	3C
disp32 <sup>2</sup>		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8 <sup>3</sup>	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	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]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	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 prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

Only one REX prefix is allowed per instruction. If used, the REX prefix byte must immediately precede the opcode byte or the escape opcode byte (0FH). 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 be immediately preceding the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and OF 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 form of INC/DEC instruction 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:

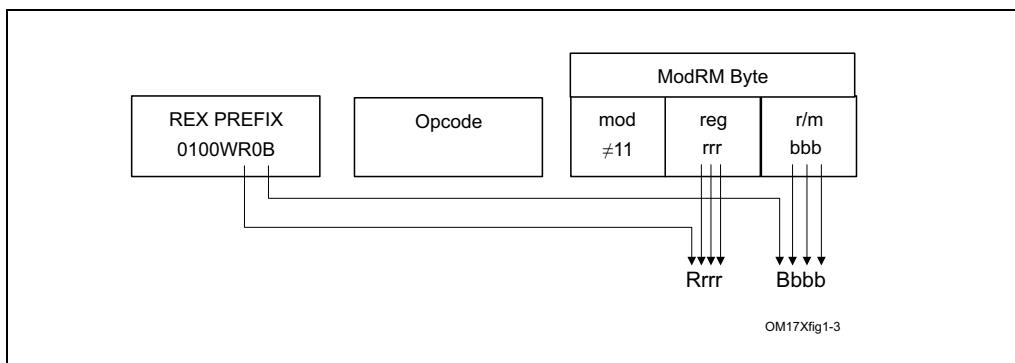
- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.

## INSTRUCTION FORMAT

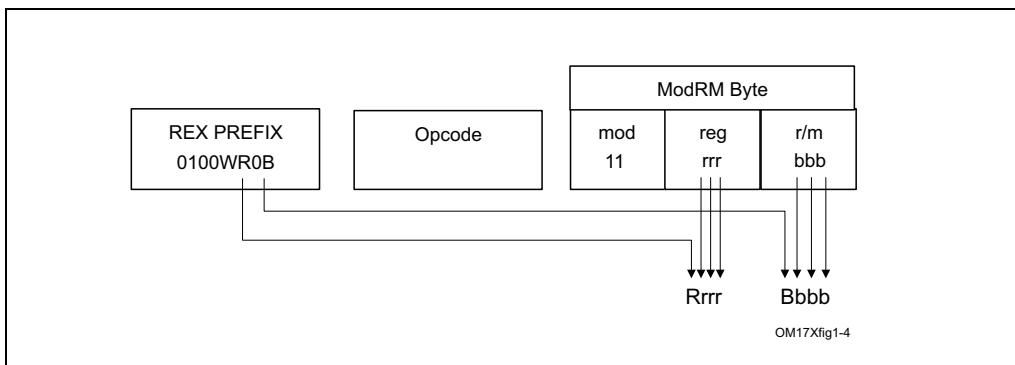
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.
- REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

**Table 2-4. REX Prefix Fields [BITS: 0100W|RXB]**

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**

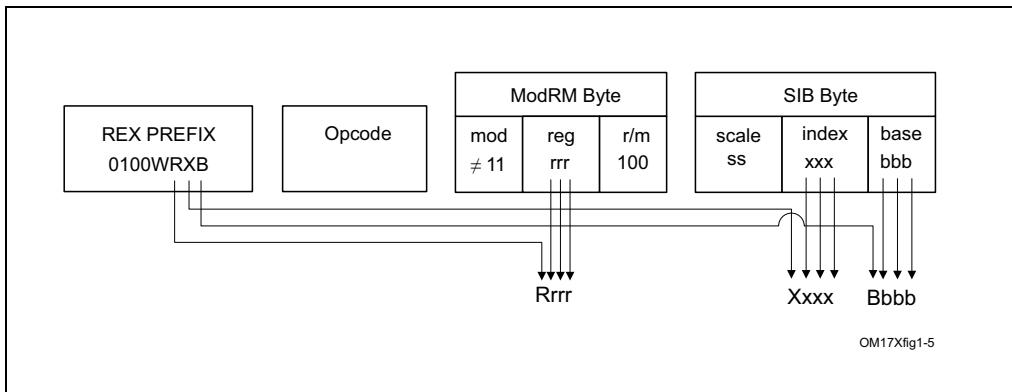


Figure 2-6. Memory Addressing With a SIB Byte

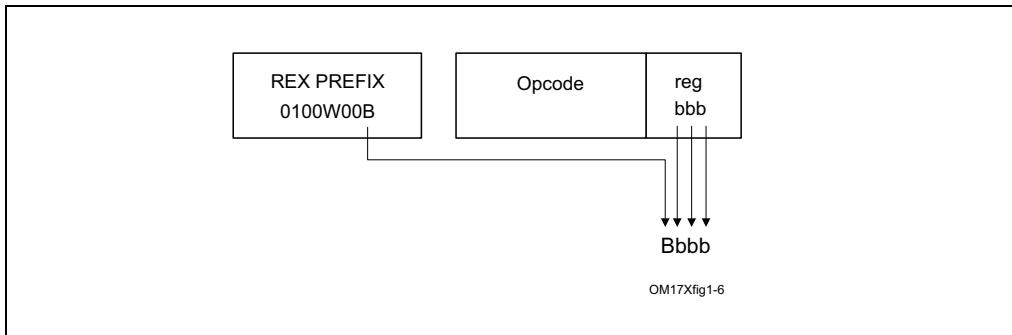


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X &amp; REX.R Not Used

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	SIB byte present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (b) which is not decoded (don't care).
	r/m = b*100(ESP)			SIB byte also required for R12-based addressing.
ModR/M Byte	mod = 0	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).
	r/m = b*101(EBP)			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.
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 instruction 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  $\pm 2\text{GB}$  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	Disp32	RIP + Disp32	Must use SIB form with normal (zero-based) displacement addressing
	r/m = 101 (none)			
SIB Byte	base = 101 (none)	if mod = 00, Disp32	Same as legacy	None
	index = 100 (none)			
	scale = 0, 1, 2, 4			

The ModR/M encoding for RIP-relative addressing does not depend on using 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 (OFH), SIMD prefix byte (66H, F2H, F3H) via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access, memory addressing, or accessing XMM8-XMM15 (including YMM8-YMM15).
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only because only a subset of SIMD instructions need them.
- Extensibility for future instruction extensions without significant instruction length increase.

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 OFH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

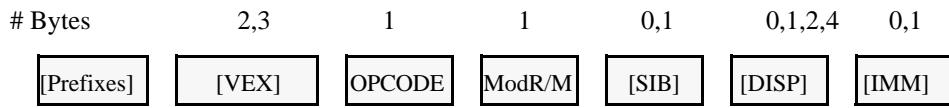


Figure 2-8. Instruction Encoding Format with VEX Prefix

### 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: OFH, and OFH 3AH/OFH 38H. The one-byte escape (OFH) and two-byte escape (OFH 3AH, OFH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, OFH, OFH 3AH, OFH 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.

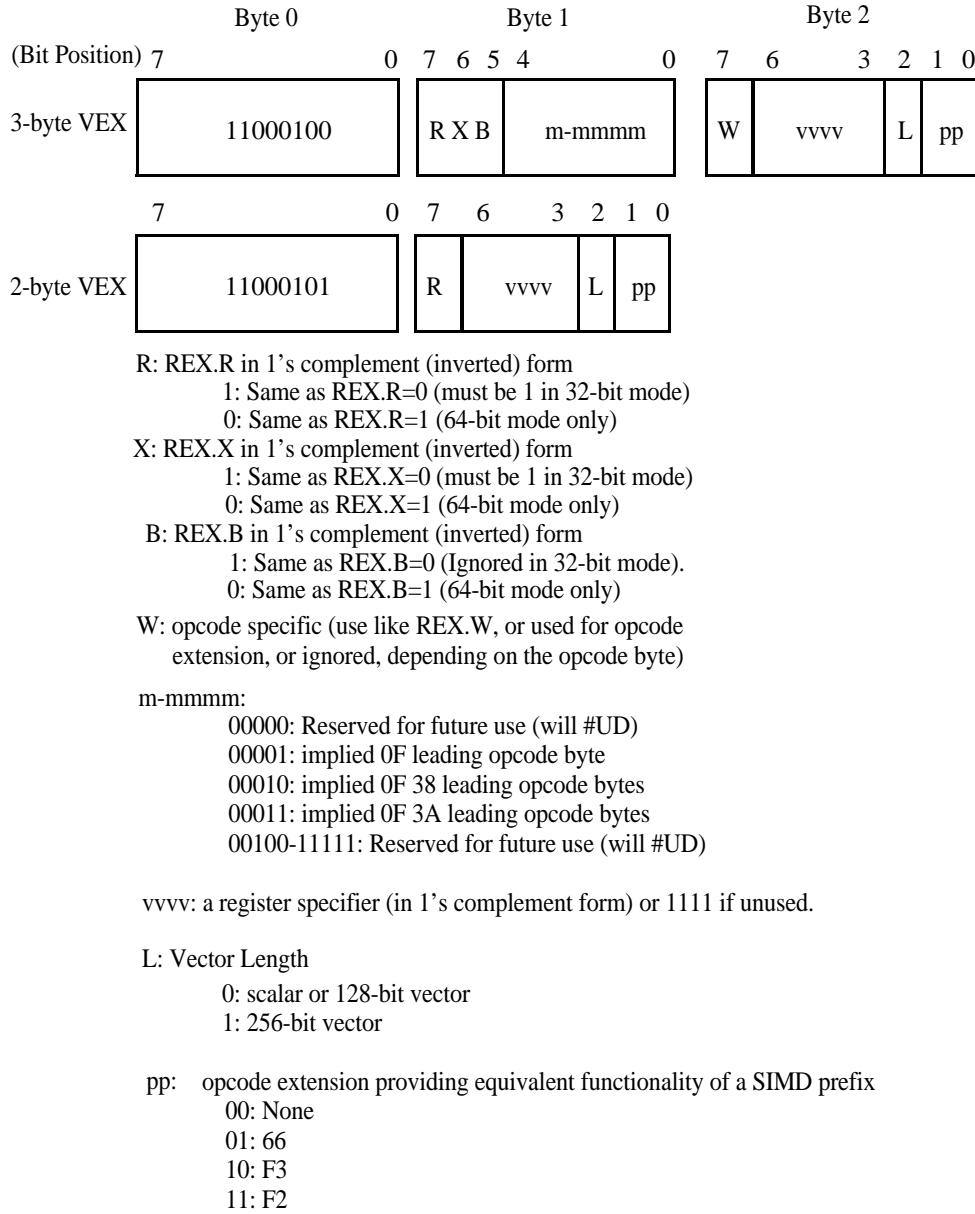
The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

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.

## INSTRUCTION FORMAT



**Figure 2-9. VEX bitfields**

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, 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.mmmmmm 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 2-8 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	Valid in Legacy/Compatibility 32-bit modes?
1111B	XMM0/YMM0	Valid
1110B	XMM1/YMM1	Valid
1101B	XMM2/YMM2	Valid
1100B	XMM3/YMM3	Valid
1011B	XMM4/YMM4	Valid
1010B	XMM5/YMM5	Valid
1001B	XMM6/YMM6	Valid
1000B	XMM7/YMM7	Valid
0111B	XMM8/YMM8	Invalid
0110B	XMM9/YMM9	Invalid
0101B	XMM10/YMM10	Invalid
0100B	XMM11/YMM11	Invalid
0011B	XMM12/YMM12	Invalid
0010B	XMM13/YMM13	Invalid
0001B	XMM14/YMM14	Invalid
0000B	XMM15/YMM15	Invalid

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

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

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

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

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

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

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves 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 (OF, OF 38, or OF 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	OF
00010B	OF 38
00011B	OF 3A
00100-11111B	Reserved
(2-byte VEX)	OF

VEX.m-mmmm is only available on the 3-byte VEX. The 2-byte VEX implies a leading OFh 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 the following table.

**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 the following table.

**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 MODRM, 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 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 XCRO.) 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 AVX Instruction Length

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

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.
- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 2-3 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. The register names also include R8L-R15L applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-3). In 32-bit mode, R8L-R15L does not apply.

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

**Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte**

r32 (In decimal) Base = (In binary) Base =			EAX/ R8L 0 000	ECX/ R9L 1 001	EDX/ R10L 2 010	EBX/ R11L 3 011	ESP/ R12L 4 100	EBP/ R13L <sup>1</sup> 5 101	ESI/ R14L 6 110	EDI/ R15L 7 111	
Scaled Index			SS	Index	Value of SIB Byte (in Hexadecimal)						
VR0/VR8	*1	00	000	00	01	02	03	04	05	06	07
VR1/VR9		001	08	09	0A	0B	0C	0D	0E	0F	
VR2/VR10		010	10	11	12	13	14	15	16	17	
VR3/VR11		011	18	19	1A	1B	1C	1D	1E	1F	
VR4/VR12		100	20	21	22	23	24	25	26	27	
VR5/VR13		101	28	29	2A	2B	2C	2D	2E	2F	
VR6/VR14		110	30	31	32	33	34	35	36	37	
VR7/VR15		111	38	39	3A	3B	3C	3D	3E	3F	
VR0/VR8	*2	01	000	40	41	42	43	44	45	46	47
VR1/VR9		001	48	49	4A	4B	4C	4D	4E	4F	
VR2/VR10		010	50	51	52	53	54	55	56	57	
VR3/VR11		011	58	59	5A	5B	5C	5D	5E	5F	
VR4/VR12		100	60	61	62	63	64	65	66	67	
VR5/VR13		101	68	69	6A	6B	6C	6D	6E	6F	
VR6/VR14		110	70	71	72	73	74	75	76	77	
VR7/VR15		111	78	79	7A	7B	7C	7D	7E	7F	

**Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte**

VR0/VR8	*4	10	000	80	81	82	83	84	85	86	87
VR1/VR9		001	88	89	8A	8B	8C	8D	8E	8F	
VR2/VR10		010	90	91	92	93	94	95	96	97	
VR3/VR11		011	98	89	9A	9B	9C	9D	9E	9F	
VR4/VR12		100	A0	A1	A2	A3	A4	A5	A6	A7	
VR5/VR13		101	A8	A9	AA	AB	AC	AD	AE	AF	
VR6/VR14		110	B0	B1	B2	B3	B4	B5	B6	B7	
VR7/VR15		111	B8	B9	BA	BB	BC	BD	BE	BF	
VR0/VR8	*8	11	000	C0	C1	C2	C3	C4	C5	C6	C7
VR1/VR9		001	C8	C9	CA	CB	CC	CD	CE	CF	
VR2/VR10		010	D0	D1	D2	D3	D4	D5	D6	D7	
VR3/VR11		011	D8	D9	DA	DB	DC	DD	DE	DF	
VR4/VR12		100	E0	E1	E2	E3	E4	E5	E6	E7	
VR5/VR13		101	E8	E9	EA	EB	EC	ED	EE	EF	
VR6/VR14		110	F0	F1	F2	F3	F4	F5	F6	F7	
VR7/VR15		111	F8	F9	FA	FB	FC	FD	FE	FF	

**NOTES:**

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

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

**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 INSTRUCTION EXCEPTION SPECIFICATION**

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table 2-14 summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.4.1 through 2.5.1. For example, ADDPS contains the entry:

*"See Exceptions Type 2"*

In this entry, "Type2" can be looked up in Table 2-14.

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 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 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	none
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)
Type 7	AVX, Legacy SSE	none	none
Type 8	AVX	none	none
Type 11	F16C	8 or 16 byte, Not explicitly aligned, no AC#	yes
Type 12	AVX2	Not explicitly aligned, no AC#	no

See Table 2-15 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)CVTPD2DQ, (V)CVTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUB132PS, VFMSUB213PS, VFMSUB231PS, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, (V)HADDPD, (V)HADDPS, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS
Type 3	(V)ADDSD, (V)ADDSS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)CVTPS2PD, (V)CVTS2SI, (V)CVTS2SS, (V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (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)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)MOV DQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPASDBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW, (V)PACKUSWB, (V)PACKUSDW, (V)PADDW, (V)PADDW, (V)PADDQ, (V)PADDDB, (V)PADDSD, (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)PMAKS B, (V)PMAKSW, (V)PMAKSD, (V)PMAKUB, (V)PMAKUW, (V)PMAKUD, (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)PSHUFBD, (V)PSHUFHD, (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)PUNPKHBW, (V)PUNPKHWD, (V)PUNPKHDQ, (V)PUNPKHQDQ, (V)PUNPKLBW, (V)PUNPKLWD, (V)PUNPKLDQ, (V)PUNPKLQDQ, (V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPKHPD, (V)UNPKHPS, (V)UNPKLPD, (V)UNPKLPS, (V)XORPD, (V)XORPS, VPBLENDD, VPERMD, VPERMPS, VPERMPD, VPERMQ, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ, VPERMILPD, VPERMILPS, VPERM2F128
Type 5	(V)CVTDQ2PD, (V)EXTRACTPS, (V)INSERTPS, (V)MOVD, (V)MOVQ, (V)MOVDDUP, (V)MOVLPD, (V)MOVLPSS, (V)MOVHPD, (V)MOVHPS, (V)MOVSD, (V)MOVSS, (V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ, (V)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX, VLDMXCSR*, VSTMXCSR
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSETRF128, VMASKMOVPS**, VMASKMOVPD**, VPMASKMOVD, VPMASKMOVQ, VBROADCASTI128, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VEXTRACTI128, VINSETI128, 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, and PCMPISTRM instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

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

**Table 2-16. #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, VPBLEND, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMILPD, VPERMILPS, VPERM2F128	
Type 5		VPEXTRQ, VPINSRQ,
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINERTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINERTI128	
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, VPCMPTB/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			

## 2.4.1 Exceptions Type 1 (Aligned memory reference)

**Table 2-18. Type 1 Class Exception Conditions**

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

## 2.4.2 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 XCRO[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.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 CRO.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.

### 2.4.3 Exceptions Type 3 (<16 Byte memory argument)

**Table 2-20. Type 3 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCRO[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.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 CRO.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

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

Table 2-21. Type 4 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.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 CRO.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. <sup>1</sup>
			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. PCMPESTRI, PCMPESTRM, PCMPISTRRI, and PCMPISTRM instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

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

**Table 2-22. Type 5 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.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 CRO.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## 2.4.6 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 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	If XCRO[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
			X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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

**Table 2-24. Type 7 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.

## 2.4.8 Exceptions Type 8 (AVX and no memory argument)

**Table 2-25. Type 8 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual 80x86 mode.
			X	X	If XCRO[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.O1H.ECX.AVX[bit 28]=0. If VEX.vvvv != 1111B.
	X	X	X	X	If proceeded by a LOCK prefix (F0H).
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.

## 2.4.9 Exception Type 11 (VEX-only, mem arg no AC, floating-point exceptions)

**Table 2-26. Type 11 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CRO.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.OSMMEXCPT[bit 10] = 1

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

**Table 2-27. Type 12 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	NA	If address size attribute is 16 bit
	X	X	X	X	If ModR/M.mod = '11b'
	X	X	X	X	If ModR/M.rm != '100b'
	X	X	X	X	If any corresponding CPUID feature flag is '0'
Device Not Available, #NM	X	X	X	X	If CRO.TS[bit 3]=1
			X		For an illegal address in the SS segment
Stack, SS(0)			X		If a memory address referencing the SS segment is in a non-canonical form
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	X	X			If the memory address is in a non-canonical form.
General Protection, #GP(0)			X		If any part of the operand lies outside the effective address space from 0 to FFFFH
	X	X			For a page fault

## 2.5 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

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

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

## INSTRUCTION FORMAT

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.

### 2.5.1 Exception Conditions for VEX-Encoded GPR Instructions

The exception conditions applicable to VEX-encoded GPR instruction differs from those of legacy GPR instructions. Table 2-28 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
See Table 2-29	ANDN, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX

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

**Table 2-29. Exception Definition (VEX-Encoded GPR Instructions)**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If BMI1/BMI2 CPUID feature flag is '0'
	X	X			If a VEX prefix is present
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
Stack, SS(0)	X	X	X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

# CHAPTER 3

## INSTRUCTION SET REFERENCE, A-M

This chapter describes the instruction set for the Intel 64 and IA-32 architectures (A-M) 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, N-Z," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

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	A	V/V	NP	Complement carry flag.

#### Instruction Operand Encoding

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

### 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:

- **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** — A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. See Table 3-1 for the codes. The +ro columns in the table are applicable only in 64-bit mode.
- **+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	REXB	Reg Field	Register	REXB	Reg Field	Register	REXB	Reg Field	Register	REXB	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	RBП	None	5
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											
R8L	Yes	0	R8W	Yes	0	R8D	Yes	0	R8	Yes	0
R9L	Yes	1	R9W	Yes	1	R9D	Yes	1	R9	Yes	1
R10L	Yes	2	R10W	Yes	2	R10D	Yes	2	R10	Yes	2
R11L	Yes	3	R11W	Yes	3	R11D	Yes	3	R11	Yes	3

**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
R12L	Yes	4	R12W	Yes	4	R12D	Yes	4	R12	Yes	4
R13L	Yes	5	R13W	Yes	5	R13D	Yes	5	R13	Yes	5
R14L	Yes	6	R14W	Yes	6	R14D	Yes	6	R14	Yes	6
R15L	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.[NDS].[128,256].[66,F2,F3].OF/OF3A/OF38.[W0,W1] opcode [/r] [/ib,/is4]**

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

- **NDS, NDD, DDS:** specifies that VEX.vvvv field is valid for the encoding of a register operand:
  - VEX.NDS: VEX.vvvv encodes the first source register in an instruction syntax where the content of source registers will be preserved.
  - VEX.NDD: VEX.vvvv encodes the destination register that cannot be encoded by ModR/M:reg field.
  - VEX.DDS: VEX.vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result.
  - If none of NDS, NDD, and DDS is present, VEX.vvvv must be 1111b (i.e. VEX.vvvv does not encode an operand). The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix.
- **128,256:** VEX.L field can be 0 (denoted by VEX.128 or VEX.LZ) or 1 (denoted by VEX.256). The VEX.L field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. The presence of the notation VEX.256 or VEX.128 in the opcode column should be interpreted as follows:
  - If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L= 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
  - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception (e.g. VMOVLPS).
  - If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distinguished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either "SS", "SD", or "SI" for SIMD floating-point conversion instructions.
  - If VEX.LZ is present in the opcode column: The VEX.L must be encoded to be 0B, an #UD occurs if VEX.L is not zero.

- **66,F2,F3:** The presence or absence of these 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.
- **OF,OF3A,OF38:** The presence maps to a valid encoding of the VEX.mmmmmm field. Only three encoded values of VEX.mmmmmm are defined as valid, corresponding to the escape byte sequence of OFH, 0F3AH and 0F38H. The effect of a valid VEX.mmmmmm 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.mmmmmm may be considered as an implied escape byte sequence of either OFH, 0F3AH or 0F38H. The VEX.mmmmmm field must be encoded using the 3-byte form of VEX prefix.
- **OF,OF3A,OF38 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 OF 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.mmmmmm) 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 imm[7:4] and instruction-specific payload in imm[3:0].
- In general, the encoding of VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 2.3.

### 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 (R8L - R15L) 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 is a signed number between –128 and +127 inclusive. 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.
- **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 R8L - R15L 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.
- **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, 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.
- **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.
- **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  $i^{\text{th}}$  element from the top of the FPU register stack ( $i \leftarrow 0$  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.
- **SRC1** — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX prefix and having two or more source operands.
- **SRC2** — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX prefix and having two or more source operands.
- **SRC3** — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX prefix and having three source operands.
- **SRC** — The source in a AVX single-source instruction or the source in a Legacy SSE instruction.
- **DST** — the destination in a AVX instruction. In Legacy SSE instructions can be either the destination, first source, or both. This field is encoded by reg\_field.

### 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.vvv field or additional operand encoding placement.

#### 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.1.ECX, CPUID.1.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 bitfield specifying a source or destination register (in 1's complement form).
- **rm\_field:** shorthand for the ModR/M *r/m* field and any REX.B
- **reg\_field:** shorthand for the ModR/M *reg* field and any REX.R

### 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  $\leftarrow$  B indicates that the value of B is assigned to A.
- The symbols =,  $\neq$ ,  $>$ ,  $<$ ,  $\geq$ , and  $\leq$  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  $\leftarrow$  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  $\leftarrow$  MOVW
  THEN OperandSize = 16;
ELSE
  IF Instruction  $\leftarrow$  MOVD
    THEN OperandSize = 32;
  ELSE
    IF Instruction  $\leftarrow$  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.
- **VLMAX** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction’s prefix but is instead determined by the current value of XCRO. For existing processors, VLMAX is 256 whenever XCRO.YMM[bit 2] is 1. Future processors may define new bits in XCRO whose setting may imply other values for VLMAX.

#### VLMAX Definition

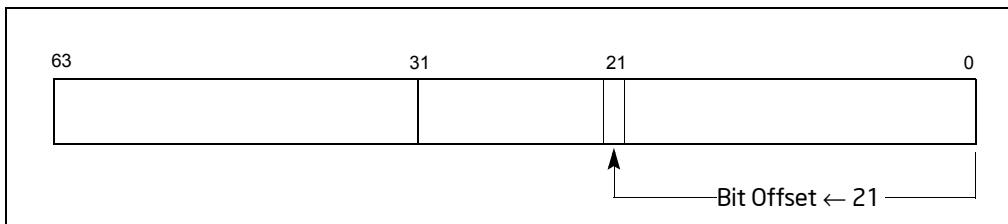
XCRO Component	VLMAX
XCRO.YMM	256

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 can range 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).

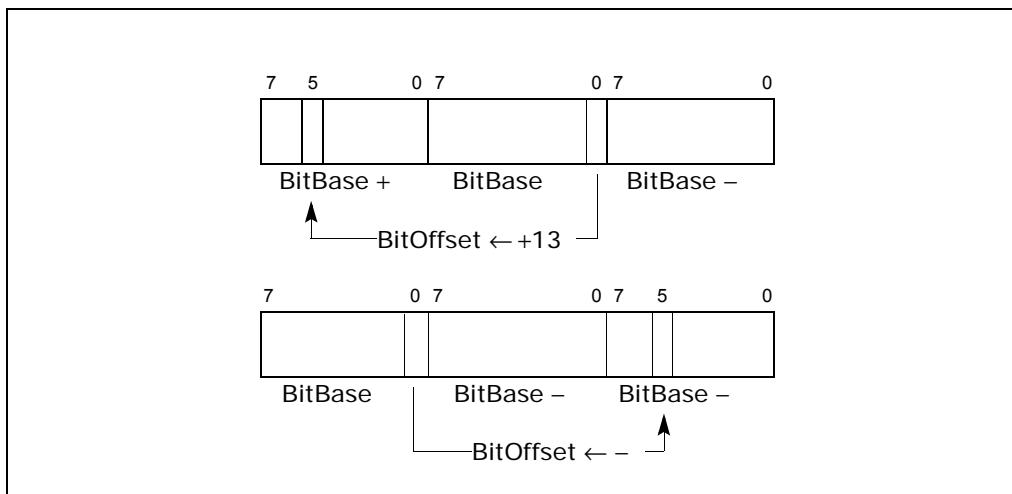


Figure 3-2. Memory Bit Indexing

### 3.1.1.10 Intel® C/C++ Compiler Intrinsic Equivalents Section

The Intel C/C++ compiler intrinsics equivalents are special C/C++ coding extensions that allow using the syntax of C function calls and C variables instead of hardware registers. Using these intrinsics frees programmers from having to manage registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that executable run faster.

The following sections discuss the intrinsics API and the MMX technology and SIMD floating-point intrinsics. Each intrinsic equivalent is listed with the instruction description. 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.

See Appendix C, "Intel® C/C++ Compiler Intrinsics and Functional Equivalents," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*, for more information on using intrinsics.

#### Intrinsics API

The benefit of coding with MMX technology intrinsics and the SSE/SSE2/SSE3 intrinsics is that you can use the syntax of C function calls and C variables instead of hardware registers. This frees you from managing registers and programming assembly. Further, the compiler optimizes the instruction scheduling so that your executable runs faster. For each computational and data manipulation instruction in the new instruction set, there is a corresponding C intrinsic that implements it directly. The intrinsics allow you to specify the underlying implementation (instruction selection) of an algorithm yet leave instruction scheduling and register allocation to the compiler.

#### MMX™ Technology Intrinsics

The MMX technology intrinsics are based on a `__m64` data type that represents the specific contents of an MMX technology register. You can specify values in bytes, short integers, 32-bit values, or a 64-bit object. The `__m64` data type, however, is not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use `__m64` data only on the left-hand side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions ("+", ">>", and so on).
- Use `__m64` objects in aggregates, such as unions to access the byte elements and structures; the address of an `__m64` object may be taken.
- Use `__m64` data only with the MMX technology intrinsics described in this manual and Intel® C/C++ compiler documentation.
- See:
  - <http://www.intel.com/support/performancetools/>

- Appendix C, “Intel® C/C++ Compiler Intrinsics and Functional Equivalents,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C*, for more information on using intrinsics.
- SSE/SSE2/SSE3 Intrinsics
- SSE/SSE2/SSE3 intrinsics all make use of the XMM registers of the Pentium III, Pentium 4, and Intel Xeon processors. There are three data types supported by these intrinsics: `__m128`, `__m128d`, and `__m128i`.
- The `__m128` data type is used to represent the contents of an XMM register used by an SSE intrinsic. This is either four packed single-precision floating-point values or a scalar single-precision floating-point value.
- The `__m128d` data type holds two packed double-precision floating-point values or a scalar double-precision floating-point value.
- The `__m128i` data type can hold sixteen byte, eight word, or four doubleword, or two quadword integer values.

The compiler aligns `__m128`, `__m128d`, and `__m128i` local and global data to 16-byte boundaries on the stack. To align integer, float, or double arrays, use the `declspec` statement as described in Intel C/C++ compiler documentation. See <http://www.intel.com/support/performancetools/>.

The `__m128`, `__m128d`, and `__m128i` data types are not basic ANSI C data types and therefore some restrictions are placed on its usage:

- Use `__m128`, `__m128d`, and `__m128i` only on the left-hand side of an assignment, as a return value, or as a parameter. Do not use it in other arithmetic expressions such as “`+`” and “`>>`.”
- Do not initialize `__m128`, `__m128d`, and `__m128i` with literals; there is no way to express 128-bit constants.
- Use `__m128`, `__m128d`, and `__m128i` objects in aggregates, such as unions (for example, to access the float elements) and structures. The address of these objects may be taken.
- Use `__m128`, `__m128d`, and `__m128i` data only with the intrinsics described in this user’s guide. See Appendix C, “Intel® C/C++ Compiler Intrinsics and Functional Equivalents,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C*, for more information on using intrinsics.

The compiler aligns `__m128`, `__m128d`, and `__m128i` local data to 16-byte boundaries on the stack. Global `__m128` data is also aligned on 16-byte boundaries. (To align float arrays, you can use the alignment `declspec` described in the following section.) Because the new instruction set treats the SIMD floating-point registers in the same way whether you are using packed or scalar data, there is no `__m32` data type to represent scalar data as you might expect. For scalar operations, you should use the `__m128` objects and the “scalar” forms of the intrinsics; the compiler and the processor implement these operations with 32-bit memory references.

The suffixes `ps` and `ss` are used to denote “packed single” and “scalar single” precision operations. The packed floats are represented in right-to-left order, with the lowest word (right-most) being used for scalar operations: `[z, y, x, w]`. To explain how memory storage reflects this, consider the following example.

The operation:

```
float a[4] ← { 1.0, 2.0, 3.0, 4.0 };
__m128 t ← _mm_load_ps(a);
```

Produces the same result as follows:

```
_m128 t ← _mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

In other words:

```
t ← [ 4.0, 3.0, 2.0, 1.0 ]
```

Where the “scalar” element is 1.0.

Some intrinsics are “composites” because they require more than one instruction to implement them. You should be familiar with the hardware features provided by the SSE, SSE2, SSE3, and MMX technology when writing programs with the intrinsics.

Keep the following important issues in mind:

- Certain intrinsics, such as `_mm_load_ps` and `_mm_cmplt_ss`, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful of their implementation cost.
- Data loaded or stored as `__m128` objects must generally be 16-byte-aligned.

- Some intrinsics require that their argument be immediates, that is, constant integers (literals), due to the nature of the instruction.
- The result of arithmetic operations acting on two NaN (Not a Number) arguments is undefined. Therefore, floating-point operations using NaN arguments may not match the expected behavior of the corresponding assembly instructions.

For a more detailed description of each intrinsic and additional information related to its usage, refer to Intel C/C++ compiler documentation. See:

- <http://www.intel.com/support/performancetools/>
- Appendix C, “Intel® C/C++ Compiler Intrinsics and Functional Equivalents,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C*, for more information on using intrinsics.

### 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 3-3 associates each two-letter mnemonic with the corresponding interrupt vector number and exception 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 3-3. Intel 64 and IA-32 General Exceptions**

Vector No.	Name	Source	Protected Mode <sup>7</sup>	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	INT 3 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)	UD2 instruction or reserved opcode.	Yes	Yes	Yes
7	#NM—Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes

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

Vector No.	Name	Source	Protected Mode <sup>1</sup>	Real Address Mode	Virtual 8086 Mode
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	Reserved	Yes
11	#NP—Segment Not Present	Loading segment registers or accessing system segments.	Yes	Reserved	Yes
12	#SS—Stack Segment Fault	Stack operations and SS register loads.	Yes	Yes	Yes
13	#GP—General Protection <sup>2</sup>	Any memory reference and other protection checks.	Yes	Yes	Yes
14	#PF—Page Fault	Any memory reference.	Yes	Reserved	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	Reserved	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

**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.

**3.1.1.14 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 3-3).

**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 3-3).

**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, vector number 16) being generated. Table 3-4 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 3-4. x87 FPU Floating-Point Exceptions**

Mnemonic	Name	Source
#IS	Floating-point invalid operation: - Stack overflow or underflow	- x87 FPU stack overflow or underflow
#IA	- Invalid arithmetic operation	- 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, vector number 19) being generated. Table 3-5 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 3-5. 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 INSTRUCTIONS (A-M)

The remainder of this chapter provides descriptions of Intel 64 and IA-32 instructions (A-M). See also: Chapter 4, “Instruction Set Reference, N-Z,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

**AAA—ASCII Adjust After Addition**

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

**Instruction Operand Encoding**

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

**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 OFH) > 9) or (AF = 1)
      THEN
        AL ← AL + 6;
        AH ← AH + 1;
        AF ← 1;
        CF ← 1;
        AL ← AL AND OFH;
      ELSE
        AF ← 0;
        CF ← 0;
        AL ← AL AND OFH;
      FI;
  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	NP	Invalid	Valid	ASCII adjust AX before division.
D5 <i>ib</i>	AAD imm8	NP	Invalid	Valid	Adjust AX before division to number base <i>imm8</i> .

**Instruction Operand Encoding**

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

**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	NP	Invalid	Valid	ASCII adjust AX after multiply.
D4 <i>ib</i>	AAM imm8	NP	Invalid	Valid	Adjust AX after multiply to number base <i>imm8</i> .

### Instruction Operand Encoding

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

### 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	NP	Invalid	Valid	ASCII adjust AL after subtraction.

**Instruction Operand Encoding**

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

**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 OFH) > 9) or (AF = 1)
      THEN
        AX ← AX - 6;
        AH ← AH - 1;
        AF ← 1;
        CF ← 1;
        AL ← AL AND OFH;
      ELSE
        CF ← 0;
        AF ← 0;
        AL ← AL AND OFH;
    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 <i>ib</i>	ADC AL, <i>imm8</i>	I	Valid	Valid	Add with carry <i>imm8</i> to AL.
15 <i>iw</i>	ADC AX, <i>imm16</i>	I	Valid	Valid	Add with carry <i>imm16</i> to AX.
15 <i>id</i>	ADC EAX, <i>imm32</i>	I	Valid	Valid	Add with carry <i>imm32</i> to EAX.
REX.W + 15 <i>id</i>	ADC RAX, <i>imm32</i>	I	Valid	N.E.	Add with carry <i>imm32</i> sign extended to 64-bits to RAX.
80 /2 <i>ib</i>	ADC r/m8, <i>imm8</i>	MI	Valid	Valid	Add with carry <i>imm8</i> to r/m8.
REX + 80 /2 <i>ib</i>	ADC r/m8*, <i>imm8</i>	MI	Valid	N.E.	Add with carry <i>imm8</i> to r/m8.
81 /2 <i>iw</i>	ADC r/m16, <i>imm16</i>	MI	Valid	Valid	Add with carry <i>imm16</i> to r/m16.
81 /2 <i>id</i>	ADC r/m32, <i>imm32</i>	MI	Valid	Valid	Add with CF <i>imm32</i> to r/m32.
REX.W + 81 /2 <i>id</i>	ADC r/m64, <i>imm32</i>	MI	Valid	N.E.	Add with CF <i>imm32</i> sign extended to 64-bits to r/m64.
83 /2 <i>ib</i>	ADC r/m16, <i>imm8</i>	MI	Valid	Valid	Add with CF sign-extended <i>imm8</i> to r/m16.
83 /2 <i>ib</i>	ADC r/m32, <i>imm8</i>	MI	Valid	Valid	Add with CF sign-extended <i>imm8</i> into r/m32.
REX.W + 83 /2 <i>ib</i>	ADC r/m64, <i>imm8</i>	MI	Valid	N.E.	Add with CF sign-extended <i>imm8</i> into r/m64.
10 /r	ADC r/m8, r8	MR	Valid	Valid	Add with carry byte register to r/m8.
REX + 10 /r	ADC r/m8*, r8*	MR	Valid	N.E.	Add with carry byte register to r/m64.
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, r/m8	RM	Valid	Valid	Add with carry r/m8 to byte register.
REX + 12 /r	ADC r8*, r/m8*	RM	Valid	N.E.	Add with carry r/m64 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:**

\*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

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

**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

$\text{DEST} \leftarrow \text{DEST} + \text{SRC} + \text{CF};$

### 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.

**ADD—Add**

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

**NOTES:**

\*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AH/EAX/RAX	imm8	NA	NA

**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 or 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

$\text{DEST} \leftarrow \text{DEST} + \text{SRC};$

## 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	CPUID Feature Flag	Description
66 0F 58 /r ADDPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Add packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 58 /r VADDPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Add packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG 58 /r VADDPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Add packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

### Instruction Operand Encoding

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

### Description

Performs a SIMD add of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of SIMD double-precision floating-point operation.

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

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

### Operation

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

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

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

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

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

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

ADDPD: `__m128d _mm_add_pd (__m128d a, __m128d b)`  
VADDPD: `__m256d _mm256_add_pd (__m256d a, __m256d b)`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2.

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

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

### Instruction Operand Encoding

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

### Description

Performs a SIMD add of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of SIMD single-precision floating-point operation.

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

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

### Operation

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

#### 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[VLMAX-1:128] ← 0
```

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

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] + \text{SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] + \text{SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] + \text{SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224]$

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

ADDPS: `__m128 _mm_add_ps(__m128 a, __m128 b)`  
 VADDPS: `__m256 _mm256_add_ps (__m256 a, __m256 b)`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2.

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

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

### Instruction Operand Encoding

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

### Description

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

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a scalar double-precision floating-point operation.

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

128-bit Legacy SSE version: Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

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

DEST[63:0]  $\leftarrow$  DEST[63:0] + SRC[63:0]  
DEST[VLMAX-1:64] (Unmodified)

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

DEST[63:0]  $\leftarrow$  SRC1[63:0] + SRC2[63:0]  
DEST[127:64]  $\leftarrow$  SRC1[127:64]  
DEST[VLMAX-1:128]  $\leftarrow$  0

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

ADDSD: \_\_m128d \_mm\_add\_sd (m128d a, m128d b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Other Exceptions

See Exceptions Type 3.

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

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 58 /r ADDSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Add the low single-precision floating-point value from <i>xmm2/m32</i> to <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.WIG 58 /r VADDSS <i>xmm1,xmm2,xmm3/m32</i>		RVM V/V	AVX	Add the low single-precision floating-point value from <i>xmm3/mem</i> to <i>xmm2</i> and store the result in <i>xmm1</i> .

**Instruction Operand Encoding**

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

**Description**

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

The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a scalar single-precision floating-point operation.

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

128-bit Legacy SSE version: Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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

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

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

```
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0
```

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

ADDSS: `__m128 _mm_add_ss(__m128 a, __m128 b)`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 3.

**ADDSUBPD—Packed Double-FP Add/Subtract**

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

**Instruction Operand Encoding**

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

**Description**

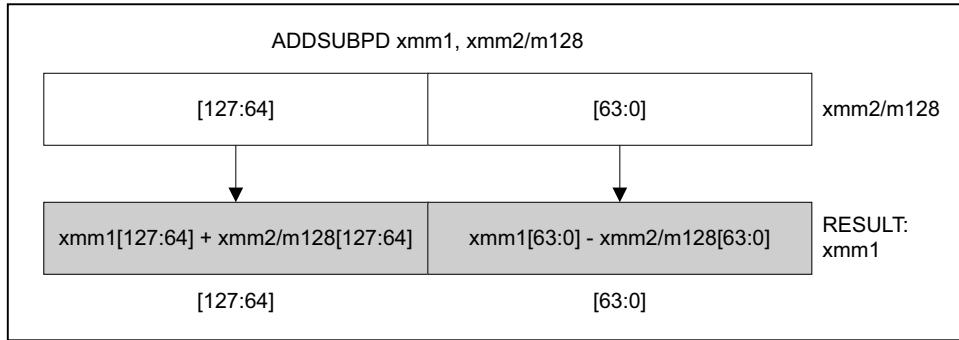
Adds odd-numbered double-precision floating-point values of the first source operand (second operand) with the corresponding double-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). 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 (VLMAX-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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

**Figure 3-3. ADDSUBPD—Packed Double-FP 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[VLMAX-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[VLMAX-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 Exceptions Type 2.

**ADDSUBPS—Packed Single-FP Add/Subtract**

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

**Instruction Operand Encoding**

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

**Description**

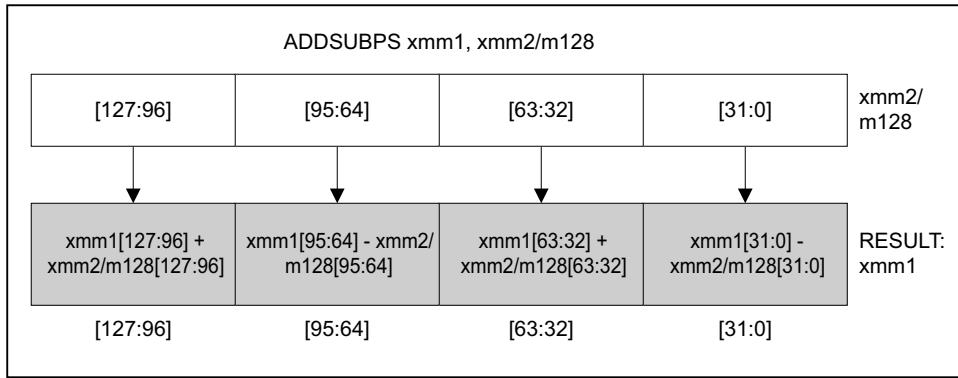
Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). 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 (VLMAX-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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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



OM15992

**Figure 3-4. ADDSUBPS—Packed Single-FP 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[VLMAX-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[VLMAX-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 Exceptions Type 2.

**AESDEC—Perform One Round of an AES Decryption Flow**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32-bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
66 0F 38 DE /r AESDEC xmm1, xmm2/m128		RM V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

**Instruction Operand Encoding**

<b>Op/En</b>	<b>Operand 1</b>	<b>Operand2</b>	<b>Operand3</b>	<b>Operand4</b>
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

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

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

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation**

```

AESDEC
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)

```

**VAESDEC**

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

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

(V)AESDEC:    \_\_m128i \_mm\_aesdec (\_\_m128i, \_\_m128i)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

**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	RM	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

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

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation****AESDECLAST**

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

**VAESDECLAST**

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

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

(V)AESDECLAST:    \_\_m128i \_mm\_aesdeclast (\_\_m128i, \_\_m128i)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

**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		RM V/V	AES	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from the xmm3/m128; store the result in xmm1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

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

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

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation****AESENC**

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

**VAESENC**

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

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

(V)AESENCL \_\_m128i \_mm\_aesenc (\_\_m128i, \_\_m128i)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

**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	RM	V/V	AES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128 bit round key from xmm3/m128; store the result in xmm1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

This instruction performs the last round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation****AESENCLAST**

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

**VAESENCLAST**

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

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

(V)AESENCLAST: \_\_m128i \_mm\_aesenclast (\_\_m128i, \_\_m128i)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

**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	Operand2	Operand3	Operand4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

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

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

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

**Operation****AESIMC**

DEST[127:0]  $\leftarrow$  InvMixColumns( SRC );  
DEST[VLMAX-1:128] (Unmodified)

**VAESIMC**

DEST[127:0]  $\leftarrow$  InvMixColumns( SRC );  
DEST[VLMAX-1:128]  $\leftarrow$  0;

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

(V)AESIMC:  $\_\_m128i\ _mm\_aesimc\ (\_\_m128i)$

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.vvvv  $\neq$  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	Operand2	Operand3	Operand4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

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

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

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[VLMAX-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[VLMAX-1:128] ← 0;

```

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

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

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.vvvv != 1111B.

## AND—Logical AND

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

### NOTES:

\*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AH/EAX/RAX	imm8	NA	NA

### 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

$\text{DEST} \leftarrow \text{DEST} \text{ AND } \text{SRC};$

## 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.NDS.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.NDS.LZ. OF38.W1 F2 /r ANDN r64a, r64b, r/m64	RVM	V/NE	BMI1	Bitwise AND of inverted r64b with r/m64, store result in r64a.

### Instruction Operand Encoding

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

### Description

Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```
DEST ← (NOT SRC1) bitwiseAND SRC2;
SF ← DEST[OperandSize - 1];
ZF ← (DEST = 0);
```

### Flags Affected

SF and ZF are updated based on result. OF and CF flags are cleared. AF and PF flags are undefined.

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

Auto-generated from high-level language.

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
#UD                    If VEX.W = 1.

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

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 54 /r ANDPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Return the bitwise logical AND of packed double-precision floating-point values in <i>xmm1</i> and <i>xmm2/m128</i> .
VEX.NDS.128.66.0F.WIG 54 /r VANDPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 54 /r VANDPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

**Description**

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

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

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

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

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

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

DEST[63:0]  $\leftarrow$  DEST[63:0] BITWISE AND SRC[63:0]  
 DEST[127:64]  $\leftarrow$  DEST[127:64] BITWISE AND SRC[127:64]  
 DEST[VLMAX-1:128] (Unmodified)

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

DEST[63:0]  $\leftarrow$  SRC1[63:0] BITWISE AND SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64] BITWISE AND SRC2[127:64]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

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

DEST[63:0]  $\leftarrow$  SRC1[63:0] BITWISE AND SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64] BITWISE AND SRC2[127:64]  
 DEST[191:128]  $\leftarrow$  SRC1[191:128] BITWISE AND SRC2[191:128]  
 DEST[255:192]  $\leftarrow$  SRC1[255:192] BITWISE AND SRC2[255:192]

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

ANDPD:    \_\_m128d \_mm\_and\_pd(\_\_m128d a, \_\_m128d b)  
VANDPD:   \_\_m256d \_mm256\_and\_pd (\_\_m256d a, \_\_m256d b)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4.

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

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 54 /r ANDPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Bitwise logical AND of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 54 /r VANDPS <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 54 /r VANDPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

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

### Description

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

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

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

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

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

### Operation

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

#### 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[VLMAX-1:128] ← 0
```

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

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE AND SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE AND SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE AND SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE AND SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] \text{ BITWISE AND SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] \text{ BITWISE AND SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] \text{ BITWISE AND SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \text{ BITWISE AND SRC2}[255:224].$

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

ANDPS:    `__m128 _mm_and_ps(__m128 a, __m128 b)`  
 VANDPS:    `__m256 _mm256_and_ps (__m256 a, __m256 b)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4.

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

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Bitwise logical AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 55 /r VANDNPD <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 55/r VANDNPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

**Description**

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

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

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

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

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

**Operation****ANDNPD (128-bit Legacy SSE version)**

DEST[63:0]  $\leftarrow$  (NOT(DEST[63:0])) BITWISE AND SRC[63:0]  
 DEST[127:64]  $\leftarrow$  (NOT(DEST[127:64])) BITWISE AND SRC[127:64]  
 DEST[VLMAX-1:128] (Unmodified)

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

DEST[63:0]  $\leftarrow$  (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

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

DEST[63:0]  $\leftarrow$  (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]  
 DEST[191:128]  $\leftarrow$  (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]  
 DEST[255:192]  $\leftarrow$  (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]

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

ANDNPD: `__m128d _mm_andnot_pd(__m128d a, __m128d b)`

VANDNPD: `__m256d _mm256_andnot_pd (__m256d a, __m256d b)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4.

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

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 55 /r ANDNPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Bitwise logical AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 55 /r VANDNPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 55 /r VANDNPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

**Description**

Inverts the bits of the four packed single-precision floating-point values in the destination operand (first operand), performs a bitwise logical AND of the four packed single-precision floating-point values in the source operand (second operand) and the temporary inverted result, and stores the result in the destination operand.

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

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

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the 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****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[VLMAX-1:128] (Unmodified)
```

**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[VLMAX-1:128] ← 0
```

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

$\text{DEST}[31:0] \leftarrow (\text{NOT}(\text{SRC1}[31:0])) \text{ BITWISE AND SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow (\text{NOT}(\text{SRC1}[63:32])) \text{ BITWISE AND SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow (\text{NOT}(\text{SRC1}[95:64])) \text{ BITWISE AND SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow (\text{NOT}(\text{SRC1}[127:96])) \text{ BITWISE AND SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow (\text{NOT}(\text{SRC1}[159:128])) \text{ BITWISE AND SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow (\text{NOT}(\text{SRC1}[191:160])) \text{ BITWISE AND SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow (\text{NOT}(\text{SRC1}[223:192])) \text{ BITWISE AND SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow (\text{NOT}(\text{SRC1}[255:224])) \text{ BITWISE AND SRC2}[255:224]$ .

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

ANDNPS:    `__m128 _mm_andnot_ps(__m128 a, __m128 b)`  
 VANDNPS:    `__m256 _mm256_andnot_ps (__m256 a, __m256 b)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4.

## ARPL—Adjust RPL Field of Segment Selector

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
63 /r	ARPL r/m16, r16	NP	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
NP	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### 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;
    Fl;
  Fl;

```

### 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.

**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 <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed DP-FP values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 0D /r ib VBLENDPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Select packed double-precision floating-point Values from <i>xmm2</i> and <i>xmm3/m128</i> from mask in <i>imm8</i> and store the values in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 0D /r ib VBLENDPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Select packed double-precision floating-point Values from <i>ymm2</i> and <i>ymm3/m256</i> from mask in <i>imm8</i> and store the values in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

VEX.256 encoded version: The first source operand is a YMM register. The second source operand 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[VLMAX-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[VLMAX-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 Exceptions Type 4.

**BEXTR – Bit Field Extract**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 -bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
VEX.NDS <sup>1</sup> .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.NDS <sup>1</sup> .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

**NOTES:**

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

**Instruction Operand Encoding**

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

**Description**

Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the 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 Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
#UD                    If VEX.W = 1.

**BLENDPS — Blend Packed Single Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A OC /r ib BLENDPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG OC /r ib VBLENDPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Select packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/m128</i> from mask in <i>imm8</i> and store the values in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG OC /r ib VBLENDPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Select packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/m256</i> from mask in <i>imm8</i> and store the values in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

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

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

**Operation****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[VLMAX-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[VLMAX-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 Exceptions Type 4.

**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 <i>xmm1, xmm2/m128 , &lt;XMM0&gt;</i>	RM0	V/V	SSE4_1	Select packed DP FP values from <i>xmm1</i> and <i>xmm2</i> from mask specified in <i>XMM0</i> and store the values in <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4B /r /is4 VBLENDVPD <i>xmm1, xmm2, xmm3/m128, xmm4</i>	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from <i>xmm2</i> or <i>xmm3/m128</i> to <i>xmm1</i> , based on mask bits in the mask operand, <i>xmm4</i> .
VEX.NDS.256.66.0F3A.W0 4B /r /is4 VBLENDVPD <i>ymm1, ymm2, ymm3/m256, ymm4</i>	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from <i>ymm2</i> or <i>ymm3/m256</i> to <i>ymm1</i> , based on mask bits in the mask operand, <i>ymm4</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute 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 (VLMAX-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 ← XMMO
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[VLMAX-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[VLMAX-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 Exceptions Type 4; 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 <i>xmm1</i> , <i>xmm2/m128</i> , < <i>XMM0</i> >	RM0	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>XMM0</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4A /r /is4 VBLENDVPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>xmm4</i>	RVMR	V/V	AVX	Conditionally copy single-precision floating-point values from <i>xmm2</i> or <i>xmm3/m128</i> to <i>xmm1</i> , based on mask bits in the specified mask operand, <i>xmm4</i> .
VEX.NDS.256.66.0F3A.W0 4A /r /is4 VBLENDVPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>ymm4</i>	RVMR	V/V	AVX	Conditionally copy single-precision floating-point values from <i>ymm2</i> or <i>ymm3/m256</i> to <i>ymm1</i> , based on mask bits in the specified mask register, <i>ymm4</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute 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 (VLMAX-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 ← XMMO
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[VLMAX-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[VLMAX-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 Exceptions Type 4; 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.NDD.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.NDD.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)	NA	NA

### Description

Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```
temp ← (-SRC) bitwiseAND (SRC);
SF ← temp[OperandSize -1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 0;
ELSE
    CF ← 1;
FI
DEST ← temp;
```

### Flags Affected

ZF and SF are updated based on the result. CF is set if the source is not zero. OF flags are cleared. AF and PF flags are undefined.

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

```
BLSI:     unsigned __int32 _blsi_u32(unsigned __int32 src);
BLSI:     unsigned __int64 _blsi_u64(unsigned __int64 src);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
#UD                    If VEX.W = 1.

**BLSMSK – Get Mask Up to Lowest Set Bit**

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /2 BLSMSK r32, r/m32	VM	V/V	BMI1	Set all lower bits in r32 to "1" starting from bit 0 to lowest set bit in r/m32.
VEX.NDD.LZ.0F38.W1 F3 /2 BLSMSK r64, r/m64	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)	NA	NA

**Description**

Sets all the lower bits of the destination operand to "1" up to and including lowest set bit (=1) in the source operand. If source operand is zero, BLSMSK sets all bits of the destination operand to 1 and also sets CF to 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

```
temp ← (SRC-1) XOR (SRC);
SF ← temp[OperandSize -1];
ZF ← 0;
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

**Flags Affected**

SF is updated based on the result. CF is set if the source is zero. ZF and OF flags are cleared. AF and PF flag are undefined.

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

BLSMSK:    unsigned \_\_int32 \_blsmask\_u32(unsigned \_\_int32 src);  
 BLSMSK:    unsigned \_\_int64 \_blsmask\_u64(unsigned \_\_int64 src);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally  
#UD              If VEX.W = 1.

## BLSR – Reset Lowest Set Bit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /1 BLSR r32, r/m32	VM	V/V	BMI1	Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.
VEX.NDD.LZ.0F38.W1 F3 /1 BLSR r64, r/m64	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)	NA	NA

### Description

Copies all bits from the source operand to the destination operand and resets (=0) the bit position in the destination operand that corresponds to the lowest set bit of the source operand. If the source operand is zero BLSR sets CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```
temp ← (SRC-1) bitwiseAND ( SRC );
SF ← temp[OperandSize -1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

### Flags Affected

ZF and SF flags are updated based on the result. CF is set if the source is zero. OF flag is cleared. AF and PF flags are undefined.

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

```
BLSR:     unsigned __int32 _blsr_u32(unsigned __int32 src);
BLSR:     unsigned __int64 _blsr_u64(unsigned __int64 src);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
#UD                    If VEX.W = 1.

## 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 $m16\&16$ .

### Instruction Operand Encoding

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

### 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)
      (* Below lower bound or above upper bound *)
      THEN #BR; Fl;
  Fl;

```

### 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
OF BC /r	BSF r16, r/m16	RM	Valid	Valid	Bit scan forward on r/m16.
OF BC /r	BSF r32, r/m32	RM	Valid	Valid	Bit scan forward on r/m32.
REX.W + OF 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)	NA	NA

**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 0, the content of the destination operand is undefined.

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
    ZF ← 1;
    DEST is undefined;
  ELSE
    ZF ← 0;
    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 all the source operand is 0; otherwise, the ZF flag is cleared. The CF, 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.

**BSR—Bit Scan Reverse**

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF BD /r	BSR r16, r/m16	RM	Valid	Valid	Bit scan reverse on r/m16.
OF BD /r	BSR r32, r/m32	RM	Valid	Valid	Bit scan reverse on r/m32.
REX.W + OF 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)	NA	NA

**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 0, the content of the destination operand is undefined.

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
    ZF ← 1;
    DEST is undefined;
  ELSE
    ZF ← 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 all the source operand is 0; otherwise, the ZF flag is cleared. The CF, 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.

## BSWAP—Byte Swap

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF C8+rd	BSWAP r32	O	Valid*	Valid	Reverses the byte order of a 32-bit register.
REX.W + OF 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
0	opcode + rd (r, w)	NA	NA	NA

### 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.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.

### 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];
    Fl;

```

### 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
OF A3 /r	BT r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag.
OF A3 /r	BT r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag.
REX.W + OF A3 /r	BT r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag.
OF BA /4 ib	BT r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag.
OF BA /4 ib	BT r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag.
REX.W + OF 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)	NA	NA
MI	ModRM:r/m (r)	imm8	NA	NA

### 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 3-10.

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:

$$\text{Effective Address} + (4 * (\text{BitOffset DIV } 32))$$

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

$$\text{Effective Address} + (2 * (\text{BitOffset DIV } 16))$$

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

$CF \leftarrow \text{Bit}(\text{BitBase}, \text{BitOffset});$

## 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
OF BB /r	BTC <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and complement.
OF BB /r	BTC <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BB /r	BTC <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and complement.
OF BA /7 <i>ib</i>	BTC <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and complement.
OF BA /7 <i>ib</i>	BTC <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BA /7 <i>ib</i>	BTC <i>r/m64, imm8</i>	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)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

### 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 3-10.

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) ← 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
OF B3 /r	BTR r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag and clear.
OF B3 /r	BTR r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + OF B3 /r	BTR r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag and clear.
OF BA /6 ib	BTR r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag and clear.
OF BA /6 ib	BTR r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + OF 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)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

### 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 3-10.

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
OF AB /r	BTS <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
OF AB /r	BTS <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF AB /r	BTS <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and set.
OF BA /5 <i>ib</i>	BTS <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
OF BA /5 <i>ib</i>	BTS <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF BA /5 <i>ib</i>	BTS <i>r/m64, imm8</i>	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)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

### 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 3-10.

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.NDS <sup>1</sup> .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.NDS <sup>1</sup> .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.

### NOTES:

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

### Instruction Operand Encoding

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

### Description

BZHI copies the bits of the first source operand (the second operand) into the destination operand (the first operand) and clears the higher bits in the destination according to the INDEX value specified by the second source operand (the third operand). The INDEX is specified by bits 7:0 of the second source operand. The INDEX value is saturated at the value of OperandSize -1. CF is set, if the number contained in the 8 low bits of the third operand is greater than OperandSize -1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```

N ← SRC2[7:0]
DEST ← SRC1
IF (N < OperandSize)
    DEST[OperandSize-1:N] ← 0
FI
IF (N > OperandSize - 1)
    CF ← 1
ELSE
    CF ← 0
FI

```

### Flags Affected

ZF, CF and SF flags are updated based on the result. OF flag is cleared. AF and PF flags are undefined.

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

```

BZHI:     unsigned __int32 _bzhi_u32(unsigned __int32 src, unsigned __int32 index);
BZHI:     unsigned __int64 _bzhi_u64(unsigned __int64 src, unsigned __int32 index);

```

### SIMD Floating-Point Exceptions

None

**Other Exceptions**

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
#UD                  If VEX.W = 1.

## CALL—Call Procedure

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
E8 cw	CALL <i>rel16</i>	M	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 cd	CALL <i>rel32</i>	M	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 <i>r/m16</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m16</i> .
FF /2	CALL <i>r/m32</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m32</i> .
FF /2	CALL <i>r/m64</i>	M	Valid	N.E.	Call near, absolute indirect, address given in <i>r/m64</i> .
9A cd	CALL <i>ptr16:16</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
9A cp	CALL <i>ptr16:32</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL <i>m16:16</i>	M	Valid	Valid	Call far, absolute indirect address given in <i>m16:16</i> .  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 <i>m16:32</i>	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 <i>m16:64</i>	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	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

### 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 7, "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 7, "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 you execute a task switch with a CALL instruction, 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 7 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 21, "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 can not 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 is 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.

## 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); Fl;
            Push(RIP);
            RIP ← tempRIP;
          Fl;
          IF OperandSize = 32
            THEN
              tempEIP ← EIP + DEST; (* DEST is rel32 *)
              IF tempEIP is not within code segment limit THEN #GP(0); Fl;
              IF stack not large enough for a 4-byte return address
                THEN #SS(0); Fl;
                Push(EIP);
                EIP ← tempEIP;
              Fl;
              IF OperandSize = 16
                THEN
                  tempEIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
                  IF tempEIP is not within code segment limit THEN #GP(0); Fl;
                  IF stack not large enough for a 2-byte return address
                    THEN #SS(0); Fl;
                    Push(IP);
                    EIP ← tempEIP;
                  Fl;
                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); Fl;
                        Push(RIP);
                        RIP ← tempRIP;
                      Fl;
                    IF OperandSize = 32
                      THEN
                        tempEIP ← DEST; (* DEST is r/m32 *)
                        IF tempEIP is not within code segment limit THEN #GP(0); Fl;
                        IF stack not large enough for a 4-byte return address
                          THEN #SS(0); Fl;
                          Push(EIP);
                          EIP ← tempEIP;

```

## INSTRUCTION SET REFERENCE, A-M

```
Fl;
IF OperandSize = 16
THEN
    tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
    IF tempEIP is not within code segment limit THEN #GP(0); Fl;
    IF stack not large enough for a 2-byte return address
        THEN #SS(0); Fl;
    Push(IP);
    EIP ← tempEIP;
Fl;
Fl;rel/abs
Fl; 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); Fl;
        IF DEST[31:16] is not zero THEN #GP(0); Fl;
        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); Fl;
        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 *)
    Fl;
Fl;

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); Fl;
    IF segment selector index not within descriptor table limits
        THEN #GP(new code segment selector); Fl;
    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); Fl;
    ELSE
        IF segment type is not a conforming or nonconforming code segment or
            64-bit call gate,
            THEN #GP(segment selector); Fl;
    Fl;
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;
Fl;

CONFORMING-CODE-SEGMENT:
IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); Fl;
IF DPL > CPL
    THEN #GP(new code segment selector); Fl;
IF segment not present
    THEN #NP(new code segment selector); Fl;
IF stack not large enough for return address
    THEN #SS(0); Fl;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN
        tempEIP ← tempEIP AND 0000FFFFH; Fl; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
    THEN #GP(0); Fl;
IF tempEIP is non-canonical
    THEN #GP(0); Fl;
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;
        Fl;
    Fl;
END;

```

```

NONCONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); Fl;

```

## INSTRUCTION SET REFERENCE, A-M

```
IF (RPL > CPL) or (DPL ≠ CPL)
    THEN #GP(new code segment selector); Fl;
IF segment not present
    THEN #NP(new code segment selector); Fl;
IF stack not large enough for return address
    THEN #SS(0); Fl;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; Fl; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
    THEN #GP(0); Fl;
IF tempEIP is non-canonical
    THEN #GP(0); Fl;
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;
        Fl;
    Fl;
END;
```

### CALL-GATE:

```
IF call gate (DPL < CPL) or (RPL > DPL)
    THEN #GP(call-gate selector); Fl;
IF call gate not present
    THEN #NP(call-gate selector); Fl;
IF call-gate code-segment selector is NULL
    THEN #GP(0); Fl;
IF call-gate code-segment selector index is outside descriptor table limits
    THEN #GP(call-gate code-segment selector); Fl;
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); Fl;
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); Fl;
IF call-gate code segment not present
    THEN #NP(call-gate code-segment selector); Fl;
IF call-gate code segment is non-conforming and DPL < CPL
    THEN go to MORE-PRIVILEGE;
    ELSE go to SAME-PRIVILEGE;
Fl;
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); Fl;
        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); Fl;
                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); Fl;
                NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
                NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        Fl;

```

```

IF IA32_EFER.LMA = 0 and NewSS is NULL
    THEN #TS(NewSS); Fl;
Read new code-segment descriptor and 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); Fl

```

```

IF IA32_EFER.LMA = 0 and new stack segment not present
    THEN #SS(NewSS); Fl;
IF CallGateSize = 32
    THEN
        IF new stack does not have room for parameters plus 16 bytes
            THEN #SS(NewSS); Fl;
        IF CallGate(InstructionPointer) not within new code-segment limit
            THEN #GP(0); Fl;
        SS ← newSS; (* Segment descriptor information also loaded *)
        ESP ← newESP;
        CS:EIP ← CallGate(CS:InstructionPointer);

```

## INSTRUCTION SET REFERENCE, A-M

```
(* 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); Fl;
      IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
        THEN #GP(0); Fl;
      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); Fl;
      IF (CallGate(InstructionPointer) is non-canonical)
        THEN #GP(0); Fl;
      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 *)
    Fl;
  Fl;
CPL ← CodeSegment(DPL)
CS(RPL) ← CPL
END;

SAME-PRIVILEGE:
IF CallGateSize = 32
  THEN
    IF stack does not have room for 8 bytes
      THEN #SS(0); Fl;
    IF CallGate(InstructionPointer) not within code segment limit
      THEN #GP(0); Fl;
    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); Fl;
        IF CallGate(InstructionPointer) not within code segment limit
          THEN #GP(0); Fl;
        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); Fl;
    IF RIP non-canonical
        THEN #GP(0); Fl;
    CS:IP ← CallGate(CS:instruction pointer);
(* Segment descriptor information also loaded *)
Push(oldCS:oldIP); (* Return address to calling procedure *)

Fl;
Fl;
CS(RPL) ← CPL
END;

TASK-GATE:
IF task gate DPL < CPL or RPL
    THEN #GP(task gate selector); Fl;
IF task gate not present
    THEN #NP(task gate selector); Fl;
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); Fl;
Access TSS descriptor in GDT;
IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
    THEN #GP(TSS selector); Fl;
IF TSS not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS (with nesting) to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
END;

TASK-STATE-SEGMENT:
IF TSS DPL < CPL or RPL
or TSS descriptor indicates TSS not available
    THEN #GP(TSS selector); Fl;
IF TSS is not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS (with nesting) to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
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.
#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.
#SS(selector)	If a memory operand effective address is outside the SS segment limit. 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, stack 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.
#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.
#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.
#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		NP	Valid	AX ← sign-extend of AL.
98	CWDE		NP	Valid	EAX ← sign-extend of AX.
REX.W + 98	CDQE		NP	Valid	N.E. RAX ← sign-extend of EAX.

### Instruction Operand Encoding

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

### 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 doubleword) 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); Fl;
  ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
    RAX ← SignExtend(EAX);
Fl;

```

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD            If the LOCK prefix is used.

**CLC—Clear Carry Flag**

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F8	CLC	NP	Valid	Valid	Clear CF flag.

**Instruction Operand Encoding**

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

**Description**

Clears the CF flag in the EFLAGS register. Operation is the same in all non-64-bit modes and 64-bit mode.

**Operation**

$CF \leftarrow 0;$

**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	NP	Valid	Valid	Clear DF flag.

**Instruction Operand Encoding**

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

**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 non-64-bit modes and 64-bit mode.

**Operation**

$DF \leftarrow 0;$

**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.

## CLFLUSH—Flush Cache Line

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF AE /7	CLFLUSH <i>m8</i>	M	Valid	Valid	Flushes cache line containing <i>m8</i> .

### Instruction Operand Encoding

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

### Description

Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see “CPUID—CPU Identification” in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions 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).

CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to ensure that previous stores are included in the write-back.

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.

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)

### 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.CLFSH[bit 19] = 0.<br>If the LOCK prefix is used.                    |

If instruction prefix is 66H, F2H or F3H.

### 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.CLFSH[bit 19] = 0.
  - If the LOCK prefix is used.
- If instruction prefix is 66H, F2H or F3H.

### 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.CLFSH[bit 19] = 0.
  - If the LOCK prefix is used.
- If instruction prefix is 66H, F2H or F3H.

## CLI – Clear Interrupt Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FA	CLI	NP	Valid	Valid	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

### Instruction Operand Encoding

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

### Description

If protected-mode virtual interrupts are not enabled, CLI clears the IF flag in the EFLAGS register. 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 affect on the generation of exceptions and NMI interrupts.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected. Table 3-6 indicates the action of the CLI instruction depending on the processor operating mode and the CPL/IOPL of the running program or procedure.

CLI operation is the same in non-64-bit modes and 64-bit mode.

**Table 3-6. Decision Table for CLI Results**

PE	VM	IOPL	CPL	PVI	VIF	VME	CLI Result
0	X	X	X	X	X	X	<b>IF = 0</b>
1	0	$\geq$ CPL	X	X	X	X	<b>IF = 0</b>
1	0	< CPL	3	1	X	X	<b>VIF = 0</b>
1	0	< CPL	< 3	X	X	X	<b>GP Fault</b>
1	0	< CPL	X	0	X	X	<b>GP Fault</b>
1	1	3	X	X	X	X	<b>IF = 0</b>
1	1	< 3	X	X	X	1	<b>VIF = 0</b>
1	1	< 3	X	X	X	0	<b>GP Fault</b>

### NOTES:

\* X = This setting has no impact.

### Operation

```

IF PE = 0
  THEN
    IF ← 0; (* Reset Interrupt Flag *)
  ELSE
    IF VM = 0;
      THEN
        IF IOPL  $\geq$  CPL
          THEN
            IF ← 0; (* Reset Interrupt Flag *)
          ELSE
            IF ((IOPL < CPL) and (CPL = 3) and (PVI = 1))
              THEN
                VIF ← 0; (* Reset Virtual Interrupt Flag *)
              ELSE

```

```

        #GP(0);
    Fl;
    Fl;
ELSE (* VM = 1 *)
    IF IOPL = 3
        THEN
            IF ← 0; (* Reset Interrupt Flag *)
        ELSE
            IF (IOPL < 3) AND (VME = 1)
                THEN
                    VIF ← 0; (* Reset Virtual Interrupt Flag *)
                ELSE
                    #GP(0);
    Fl;
    Fl;
Fl;

```

### Flags Affected

If protected-mode virtual interrupts are not enabled, IF is set to 0 if the CPL is equal to or less than the IOPL; otherwise, it is not affected. The other flags in the EFLAGS register are unaffected.

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

### Protected Mode Exceptions

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

### Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

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

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

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

## CLTS—Clear Task-Switched Flag in CRO

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF 06	CLTS	NP	Valid	Valid	Clears TS flag in CRO.

### Instruction Operand Encoding

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

### Description

Clears the task-switched (TS) flag in the CRO 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 25, “VMX Non-Root Operation,” 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

CRO.TS[bit 3] ← 0;

### Flags Affected

The TS flag in CRO 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.

**CMC—Complement Carry Flag**

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F5	CMC	NP	Valid	Valid	Complement CF flag.

**Instruction Operand Encoding**

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

**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.

## CMOVcc—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.	NA	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
OF 47 /r	CMOVNBE r32, r/m32	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
REX.W + OF 47 /r	CMOVNBE r64, r/m64	RM	Valid	N.E.	Move if not below or equal (CF=0 and ZF=0).
OF 43 /r	CMOVNC r16, r/m16	RM	Valid	Valid	Move if not carry (CF=0).
OF 43 /r	CMOVNC r32, r/m32	RM	Valid	Valid	Move if not carry (CF=0).
REX.W + OF 43 /r	CMOVNC r64, r/m64	RM	Valid	N.E.	Move if not carry (CF=0).
OF 45 /r	CMOVNE r16, r/m16	RM	Valid	Valid	Move if not equal (ZF=0).
OF 45 /r	CMOVNE r32, r/m32	RM	Valid	Valid	Move if not equal (ZF=0).
REX.W + OF 45 /r	CMOVNE r64, r/m64	RM	Valid	N.E.	Move if not equal (ZF=0).
OF 4E /r	CMOVNG r16, r/m16	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
OF 4E /r	CMOVNG r32, r/m32	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
REX.W + OF 4E /r	CMOVNG r64, r/m64	RM	Valid	N.E.	Move if not greater (ZF=1 or SF≠ OF).
OF 4C /r	CMOVNGE r16, r/m16	RM	Valid	Valid	Move if not greater or equal (SF≠ OF).
OF 4C /r	CMOVNGE r32, r/m32	RM	Valid	Valid	Move if not greater or equal (SF≠ OF).
REX.W + OF 4C /r	CMOVNGE r64, r/m64	RM	Valid	N.E.	Move if not greater or equal (SF≠ OF).
OF 4D /r	CMOVNL r16, r/m16	RM	Valid	Valid	Move if not less (SF=OF).
OF 4D /r	CMOVNL r32, r/m32	RM	Valid	Valid	Move if not less (SF=OF).
REX.W + OF 4D /r	CMOVNL r64, r/m64	RM	Valid	N.E.	Move if not less (SF=OF).
OF 4F /r	CMOVNLE r16, r/m16	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
OF 4F /r	CMOVNLE r32, r/m32	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
REX.W + OF 4F /r	CMOVNLE r64, r/m64	RM	Valid	N.E.	Move if not less or equal (ZF=0 and SF=OF).
OF 41 /r	CMOVNO r16, r/m16	RM	Valid	Valid	Move if not overflow (OF=0).
OF 41 /r	CMOVNO r32, r/m32	RM	Valid	Valid	Move if not overflow (OF=0).
REX.W + OF 41 /r	CMOVNO r64, r/m64	RM	Valid	N.E.	Move if not overflow (OF=0).
OF 4B /r	CMOVNP r16, r/m16	RM	Valid	Valid	Move if not parity (PF=0).
OF 4B /r	CMOVNP r32, r/m32	RM	Valid	Valid	Move if not parity (PF=0).
REX.W + OF 4B /r	CMOVNP r64, r/m64	RM	Valid	N.E.	Move if not parity (PF=0).
OF 49 /r	CMOVNS r16, r/m16	RM	Valid	Valid	Move if not sign (SF=0).
OF 49 /r	CMOVNS r32, r/m32	RM	Valid	Valid	Move if not sign (SF=0).
REX.W + OF 49 /r	CMOVNS r64, r/m64	RM	Valid	N.E.	Move if not sign (SF=0).
OF 45 /r	CMOVNZ r16, r/m16	RM	Valid	Valid	Move if not zero (ZF=0).
OF 45 /r	CMOVNZ r32, r/m32	RM	Valid	Valid	Move if not zero (ZF=0).
REX.W + OF 45 /r	CMOVNZ r64, r/m64	RM	Valid	N.E.	Move if not zero (ZF=0).
OF 40 /r	CMOVO r16, r/m16	RM	Valid	Valid	Move if overflow (OF=1).
OF 40 /r	CMOVO r32, r/m32	RM	Valid	Valid	Move if overflow (OF=1).
REX.W + OF 40 /r	CMOVO r64, r/m64	RM	Valid	N.E.	Move if overflow (OF=1).
OF 4A /r	CMOVP r16, r/m16	RM	Valid	Valid	Move if parity (PF=1).
OF 4A /r	CMOVP r32, r/m32	RM	Valid	Valid	Move if parity (PF=1).
REX.W + OF 4A /r	CMOVP r64, r/m64	RM	Valid	N.E.	Move if parity (PF=1).
OF 4A /r	CMOVPE r16, r/m16	RM	Valid	Valid	Move if parity even (PF=1).
OF 4A /r	CMOVPE r32, r/m32	RM	Valid	Valid	Move if parity even (PF=1).
REX.W + OF 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
OF 4B /r	CMOVPO r16, r/m16	RM	Valid	Valid	Move if parity odd (PF=0).
OF 4B /r	CMOVPO r32, r/m32	RM	Valid	Valid	Move if parity odd (PF=0).
REX.W + OF 4B /r	CMOVPO r64, r/m64	RM	Valid	N.E.	Move if parity odd (PF=0).
OF 48 /r	CMOVS r16, r/m16	RM	Valid	Valid	Move if sign (SF=1).
OF 48 /r	CMOVS r32, r/m32	RM	Valid	Valid	Move if sign (SF=1).
REX.W + OF 48 /r	CMOVS r64, r/m64	RM	Valid	N.E.	Move if sign (SF=1).
OF 44 /r	CMOVZ r16, r/m16	RM	Valid	Valid	Move if zero (ZF=1).
OF 44 /r	CMOVZ r32, r/m32	RM	Valid	Valid	Move if zero (ZF=1).
REX.W + OF 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)	NA	NA

### Description

The CMOVcc instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags 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 CMOVcc instruction.

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 CMOVcc 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 OF 47H.

The CMOVcc instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOVcc 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;
  FI;
ELSE
  IF (OperandSize = 32 and IA-32e mode active)
    THEN
      DEST[63:32] ← 0;
    FI;

```

Fl;

**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 <i>ib</i>	CMP AL, <i>imm8</i>	I	Valid	Valid	Compare <i>imm8</i> with AL.
3D <i>iw</i>	CMP AX, <i>imm16</i>	I	Valid	Valid	Compare <i>imm16</i> with AX.
3D <i>id</i>	CMP EAX, <i>imm32</i>	I	Valid	Valid	Compare <i>imm32</i> with EAX.
REX.W + 3D <i>id</i>	CMP RAX, <i>imm32</i>	I	Valid	N.E.	Compare <i>imm32 sign-extended to 64-bits</i> with RAX.
80 /7 <i>ib</i>	CMP r/m8, <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with r/m8.
REX + 80 /7 <i>ib</i>	CMP r/m8*, <i>imm8</i>	MI	Valid	N.E.	Compare <i>imm8</i> with r/m8.
81 /7 <i>iw</i>	CMP r/m16, <i>imm16</i>	MI	Valid	Valid	Compare <i>imm16</i> with r/m16.
81 /7 <i>id</i>	CMP r/m32, <i>imm32</i>	MI	Valid	Valid	Compare <i>imm32</i> with r/m32.
REX.W + 81 /7 <i>id</i>	CMP r/m64, <i>imm32</i>	MI	Valid	N.E.	Compare <i>imm32 sign-extended to 64-bits</i> with r/m64.
83 /7 <i>ib</i>	CMP r/m16, <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with r/m16.
83 /7 <i>ib</i>	CMP r/m32, <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with r/m32.
REX.W + 83 /7 <i>ib</i>	CMP r/m64, <i>imm8</i>	MI	Valid	N.E.	Compare <i>imm8</i> with r/m64.
38 /r	CMP r/m8, r8	MR	Valid	Valid	Compare r8 with r/m8.
REX + 38 /r	CMP r/m8*, r8*	MR	Valid	N.E.	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, r/m8	RM	Valid	Valid	Compare r/m8 with r8.
REX + 3A /r	CMP r8*, r/m8*	RM	Valid	N.E.	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:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (w)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AE/X/A/RAX	imm8	NA	NA

### 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 Jcc, CMOVcc, and SETcc 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.

## CMPPD—Compare Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Compare packed double-precision floating-point values in <i>xmm2/m128</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.128.66.0F.WIG C2 /r ib VCMPPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Compare packed double-precision floating-point values in <i>xmm3/m128</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.
VEX.NDS.256.66.0F.WIG C2 /r ib VCMPPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Compare packed double-precision floating-point values in <i>ymm3/m256</i> and <i>ymm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Performs a SIMD compare of the packed double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to +0.0.

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. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand.

**Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions**

Predic- cate	imm8 Encoding	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Oper-and Signals Invalid
EQ	000B	Equal	A = B		False	No
LT	001B	Less-than	A < B		False	Yes
LE	010B	Less-than-or-equal	A ≤ B		False	Yes
		Greater than	A > B	Swap Operands, Use LT	False	Yes
		Greater-than-or-equal	A ≥ B	Swap Operands, Use LE	False	Yes
UNORD	011B	Unordered	A, B = Unordered		True	No
NEQ	100B	Not-equal	A ≠ B		True	No
NLT	101B	Not-less-than	NOT(A < B)		True	Yes

**Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)**

Predicate	imm8 Encoding	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Oper-and Signals Invalid
NLE	110B	Not-less-than-or-equal	NOT(A ≤ B)		True	Yes
		Not-greater-than	NOT(A > B)	Swap Operands, Use NLT	True	Yes
		Not-greater-than-or-equal	NOT(A ≥ B)	Swap Operands, Use NLE	True	Yes
ORD	111B	Ordered	A, B = Ordered		False	No

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 the processors with "CPUID.1H:ECX.AVX =0" do not implement the greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-8. Pseudo-Op and CMPPD Implementation**

Pseudo-Op	CMPPD Implementation
CMPEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 0</i>
CMLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 1</i>
CMPLEPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 2</i>
CMPUNORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 3</i>
CMPNEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 4</i>
CMPNLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 5</i>
CMPNLEPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 6</i>
CMPORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement, require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

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

#### Enhanced Comparison Predicate for VEX-Encoded VCMPPD

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 (VLMAX-1:128) of the destination YMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

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 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 3-9). Bits 5 through 7 of the immediate are reserved.

**Table 3-9. Comparison Predicate for VCMPPD and VCMPPS Instructions**

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN <sup>1</sup>
			A > B	A < B	A = B	Unordered <sup>1</sup>	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, non-signaling)	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 3-9. Comparison Predicate for VCMPPD and VCMPPS 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 <sup>1</sup>	
NGE_UQ	19H	Not-greater-than-or-equal (unordered, nonsignaling)	False	True	False	True	No
NGT_UQ	1AH	Not-greater-than (unordered, non-signaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

**NOTES:**

1. If either operand A or B is a NAN.

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-10, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

**Table 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

**Table 3-10. Pseudo-Op and VCMPPD Implementation**

Pseudo-Op	CMPPD Implementation
VCMPUNORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 13H</i>
VCMPNEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 14H</i>
VCMPNLT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 15H</i>
VCMPNLE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 16H</i>
VCMPORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1AH</i>
VCMPFALSE_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1BH</i>
VCMPNEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1CH</i>
VCMPGE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1DH</i>
VCMPGT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1EH</i>
VCMPTRUE_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1FH</i>

## Operation

CASE (COMPARISON PREDICATE) OF

0: OP3  $\leftarrow$  EQ\_OQ; OP5  $\leftarrow$  EQ\_OQ;  
 1: OP3  $\leftarrow$  LT\_OS; OP5  $\leftarrow$  LT\_OS;  
 2: OP3  $\leftarrow$  LE\_OS; OP5  $\leftarrow$  LE\_OS;  
 3: OP3  $\leftarrow$  UNORD\_Q; OP5  $\leftarrow$  UNORD\_Q;  
 4: OP3  $\leftarrow$  NEQ\_UQ; OP5  $\leftarrow$  NEQ\_UQ;  
 5: OP3  $\leftarrow$  NLT\_US; OP5  $\leftarrow$  NLT\_US;  
 6: OP3  $\leftarrow$  NLE\_US; OP5  $\leftarrow$  NLE\_US;  
 7: OP3  $\leftarrow$  ORD\_Q; OP5  $\leftarrow$  ORD\_Q;  
 8: OP5  $\leftarrow$  EQ\_UQ;  
 9: OP5  $\leftarrow$  NGE\_US;  
 10: OP5  $\leftarrow$  NGT\_US;  
 11: OP5  $\leftarrow$  FALSE\_OQ;  
 12: OP5  $\leftarrow$  NEQ\_OQ;  
 13: OP5  $\leftarrow$  GE\_OS;  
 14: OP5  $\leftarrow$  GT\_OS;  
 15: OP5  $\leftarrow$  TRUE\_UQ;  
 16: OP5  $\leftarrow$  EQ\_OS;  
 17: OP5  $\leftarrow$  LT\_OQ;  
 18: OP5  $\leftarrow$  LE\_OQ;  
 19: OP5  $\leftarrow$  UNORD\_S;  
 20: OP5  $\leftarrow$  NEQ\_US;  
 21: OP5  $\leftarrow$  NLT\_UQ;  
 22: OP5  $\leftarrow$  NLE\_UQ;  
 23: OP5  $\leftarrow$  ORD\_S;  
 24: OP5  $\leftarrow$  EQ\_US;  
 25: OP5  $\leftarrow$  NGE\_UQ;  
 26: OP5  $\leftarrow$  NGT\_UQ;  
 27: OP5  $\leftarrow$  FALSE\_OS;  
 28: OP5  $\leftarrow$  NEQ\_OS;  
 29: OP5  $\leftarrow$  GE\_OQ;

30: OP5  $\leftarrow$  GT\_OQ;  
 31: OP5  $\leftarrow$  TRUE\_US;  
 DEFAULT: Reserved;

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

CMP0  $\leftarrow$  SRC1[63:0] OP3 SRC2[63:0];  
 CMP1  $\leftarrow$  SRC1[127:64] OP3 SRC2[127:64];  
 IF CMP0 = TRUE  
   THEN DEST[63:0]  $\leftarrow$  FFFFFFFFFFFFFFFFH;  
   ELSE DEST[63:0]  $\leftarrow$  0000000000000000H; FI;  
 IF CMP1 = TRUE  
   THEN DEST[127:64]  $\leftarrow$  FFFFFFFFFFFFFFFFH;  
   ELSE DEST[127:64]  $\leftarrow$  0000000000000000H; FI;  
 DEST[VLMAX-1:128] (Unmodified)

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

CMP0  $\leftarrow$  SRC1[63:0] OP5 SRC2[63:0];  
 CMP1  $\leftarrow$  SRC1[127:64] OP5 SRC2[127:64];  
 IF CMP0 = TRUE  
   THEN DEST[63:0]  $\leftarrow$  FFFFFFFFFFFFFFFFH;  
   ELSE DEST[63:0]  $\leftarrow$  0000000000000000H; FI;  
 IF CMP1 = TRUE  
   THEN DEST[127:64]  $\leftarrow$  FFFFFFFFFFFFFFFFH;  
   ELSE DEST[127:64]  $\leftarrow$  0000000000000000H; FI;  
 DEST[VLMAX-1:128]  $\leftarrow$  0

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

CMP0  $\leftarrow$  SRC1[63:0] OP5 SRC2[63:0];  
 CMP1  $\leftarrow$  SRC1[127:64] OP5 SRC2[127:64];  
 CMP2  $\leftarrow$  SRC1[191:128] OP5 SRC2[191:128];  
 CMP3  $\leftarrow$  SRC1[255:192] OP5 SRC2[255:192];  
 IF CMP0 = TRUE  
   THEN DEST[63:0]  $\leftarrow$  FFFFFFFFFFFFFFFFH;  
   ELSE DEST[63:0]  $\leftarrow$  0000000000000000H; FI;  
 IF CMP1 = TRUE  
   THEN DEST[127:64]  $\leftarrow$  FFFFFFFFFFFFFFFFH;  
   ELSE DEST[127:64]  $\leftarrow$  0000000000000000H; FI;  
 IF CMP2 = TRUE  
   THEN DEST[191:128]  $\leftarrow$  FFFFFFFFFFFFFFFFH;  
   ELSE DEST[191:128]  $\leftarrow$  0000000000000000H; FI;  
 IF CMP3 = TRUE  
   THEN DEST[255:192]  $\leftarrow$  FFFFFFFFFFFFFFFFH;  
   ELSE DEST[255:192]  $\leftarrow$  0000000000000000H; FI;

**Intel C/C++ Compiler Intrinsic Equivalents**

CMPPD for equality:  $\text{__m128d}_\text{mm\_cmpeq\_pd}(\text{__m128d } a, \text{__m128d } b)$   
 CMPPD for less-than:  $\text{__m128d}_\text{mm\_cmplt\_pd}(\text{__m128d } a, \text{__m128d } b)$   
 CMPPD for less-than-or-equal:  $\text{__m128d}_\text{mm\_cmple\_pd}(\text{__m128d } a, \text{__m128d } b)$   
 CMPPD for greater-than:  $\text{__m128d}_\text{mm\_cmpgt\_pd}(\text{__m128d } a, \text{__m128d } b)$   
 CMPPD for greater-than-or-equal:  $\text{__m128d}_\text{mm\_cmpge\_pd}(\text{__m128d } a, \text{__m128d } b)$   
 CMPPD for inequality:  $\text{__m128d}_\text{mm\_cmpneq\_pd}(\text{__m128d } a, \text{__m128d } b)$   
 CMPPD for not-less-than:  $\text{__m128d}_\text{mm\_cmpnlt\_pd}(\text{__m128d } a, \text{__m128d } b)$

CMPPD for not-greater-than: `__m128d _mm_cmppgt_pd(__m128d a, __m128d b)`  
CMPPD for not-greater-than-or-equal: `__m128d _mm_cmppnge_pd(__m128d a, __m128d b)`  
CMPPD for ordered: `__m128d _mm_cmppord_pd(__m128d a, __m128d b)`  
CMPPD for unordered: `__m128d _mm_cmppunord_pd(__m128d a, __m128d b)`  
CMPPD for not-less-than-or-equal: `__m128d _mm_cmppnle_pd(__m128d a, __m128d b)`  
VCMPPD: `__m256 _mm256_cmpp_pd(__m256 a, __m256 b, const int imm)`  
VCMPPD: `__m128 _mm_cmpp_pd(__m128 a, __m128 b, const int imm)`

### SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in above table, Denormal.

### Other Exceptions

See Exceptions Type 2.

**CMPPS—Compare Packed Single-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
OF C2 /r ib CMPPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE	Compare packed single-precision floating-point values in <i>xmm2/mem</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.128.0F.WIG C2 /r ib VCMPPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Compare packed single-precision floating-point values in <i>xmm3/m128</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.
VEX.NDS.256.0F.WIG C2 /r ib VCMPPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Compare packed single-precision floating-point values in <i>ymm3/m256</i> and <i>ymm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

Performs a SIMD compare of the packed single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

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. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand.

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 a fault, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX =0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-11. Compiler should treat reserved Imm8 values as illegal syntax.

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

**Table 3-11. Pseudo-Ops and CMPPS**

Pseudo-Op	Implementation
CMPEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 0</i>
CMPLETPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 1</i>
CMPLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 2</i>
CMPUNORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 3</i>
CMPNEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 4</i>
CMPNLTPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 5</i>
CMPNLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 6</i>
CMPORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 7</i>

The greater-than relations not implemented by processor 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.)

#### Enhanced Comparison Predicate for VEX-Encoded VCMPPS

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 (VLMAX-1:128) of the destination YMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

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 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 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-9, 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. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

**Table 3-12. Pseudo-Op and VCMPPS Implementation**

Pseudo-Op	CMPPS Implementation
VCMPEQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1</i>
VCMPLEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 2</i>
VCMPPUNORDPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 3</i>
VCMPEQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 4</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 5</i>
VCMLEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 6</i>
VCMPPORDPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 8</i>
VCMPEQEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 9</i>
VCMPPNGTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, OAH</i>
VCMPPFALSEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, OBH</i>

**Table 3-12. Pseudo-Op and VCMPPS Implementation**

Pseudo-Op	CMPPS Implementation
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  $\leftarrow$  EQ\_OQ; OP5  $\leftarrow$  EQ\_OQ;
- 1: OP3  $\leftarrow$  LT\_OS; OP5  $\leftarrow$  LT\_OS;
- 2: OP3  $\leftarrow$  LE\_OS; OP5  $\leftarrow$  LE\_OS;
- 3: OP3  $\leftarrow$  UNORD\_Q; OP5  $\leftarrow$  UNORD\_Q;
- 4: OP3  $\leftarrow$  NEQ\_UQ; OP5  $\leftarrow$  NEQ\_UQ;
- 5: OP3  $\leftarrow$  NLT\_US; OP5  $\leftarrow$  NLT\_US;
- 6: OP3  $\leftarrow$  NLE\_US; OP5  $\leftarrow$  NLE\_US;
- 7: OP3  $\leftarrow$  ORD\_Q; OP5  $\leftarrow$  ORD\_Q;
- 8: OP5  $\leftarrow$  EQ\_UQ;
- 9: OP5  $\leftarrow$  NGE\_US;
- 10: OP5  $\leftarrow$  NGT\_US;
- 11: OP5  $\leftarrow$  FALSE\_OQ;
- 12: OP5  $\leftarrow$  NEQ\_OQ;
- 13: OP5  $\leftarrow$  GE\_OS;
- 14: OP5  $\leftarrow$  GT\_OS;
- 15: OP5  $\leftarrow$  TRUE\_UQ;
- 16: OP5  $\leftarrow$  EQ\_OS;
- 17: OP5  $\leftarrow$  LT\_OQ;
- 18: OP5  $\leftarrow$  LE\_OQ;
- 19: OP5  $\leftarrow$  UNORD\_S;
- 20: OP5  $\leftarrow$  NEQ\_US;

```

21: OP5 ← 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
EASC;

```

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

```

CMP0 ← SRC1[31:0] OP3 SRC2[31:0];
CMP1 ← SRC1[63:32] OP3 SRC2[63:32];
CMP2 ← SRC1[95:64] OP3 SRC2[95:64];
CMP3 ← SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 00000000H; Fl;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 00000000H; Fl;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 00000000H; Fl;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 00000000H; Fl;
DEST[VLMAX-1:128] (Unmodified)

```

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

```

CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 00000000H; Fl;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 00000000H; Fl;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 00000000H; Fl;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 00000000H; Fl;
DEST[VLMAX-1:128] ← 0

```

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

```

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

```

**Intel C/C++ Compiler Intrinsic Equivalents**

CMPPS for equality: `__m128_mm_cmpeq_ps(__m128 a, __m128 b)`  
 CMPPS for less-than: `__m128_mm_cmplt_ps(__m128 a, __m128 b)`  
 CMPPS for less-than-or-equal: `__m128_mm_cmple_ps(__m128 a, __m128 b)`  
 CMPPS for greater-than: `__m128_mm_cmpgt_ps(__m128 a, __m128 b)`  
 CMPPS for greater-than-or-equal: `__m128_mm_cmpge_ps(__m128 a, __m128 b)`  
 CMPPS for inequality: `__m128_mm_cmpneq_ps(__m128 a, __m128 b)`  
 CMPPS for not-less-than: `__m128_mm_cmpnlt_ps(__m128 a, __m128 b)`  
 CMPPS for not-greater-than: `__m128_mm_cmpngt_ps(__m128 a, __m128 b)`  
 CMPPS for not-greater-than-or-equal: `__m128_mm_cmpnge_ps(__m128 a, __m128 b)`  
 CMPPS for ordered: `__m128_mm_cmpord_ps(__m128 a, __m128 b)`  
 CMPPS for unordered: `__m128_mm_cmpunord_ps(__m128 a, __m128 b)`  
 CMPPS for not-less-than-or-equal: `__m128_mm_cmplne_ps(__m128 a, __m128 b)`  
 VCMPPS: `__m256_mm256_cmp_ps(__m256 a, __m256 b, const int imm)`  
 VCMPPS: `__m128_mm_cmp_ps(__m128 a, __m128 b, const int imm)`

## SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in above table, Denormal.

## Other Exceptions

See Exceptions Type 2.

**CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands**

<b>Opcode</b>	<b>Instruction</b>	<b>Op/ En</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
A6	CMPS <i>m8, m8</i>	NP	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 <i>m16, m16</i>	NP	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 <i>m32, m32</i>	NP	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 <i>m64, m64</i>	NP	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	NP	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	NP	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	NP	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	NP	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
NP	NA	NA	NA	NA

**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 location. 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 REP prefix for block comparisons. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See “REP/REPE/REPZ /REPNE/REPNZ—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.

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;
    Fl;
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;
    Fl;
Fl;
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;
            Fl;
        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;
            Fl;
        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;
            Fl;
        Fl;
    Fl;

```

### 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 Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF C2 /r ib CMPSD <i>xmm1, xmm2/m64, imm8</i>	RMI	V/V	SSE2	Compare low double-precision floating-point value in <i>xmm2/m64</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.LIG.F2.0F.WIG C2 /r ib VCMPSD <i>xmm1, xmm2, xmm3/m64, imm8</i>	RVMI	V/V	AVX	Compare low double precision floating-point value in <i>xmm3/m64</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as comparison predicate.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

Compares the low double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a quad-word mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

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. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

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 a fault, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX =0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-13. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-13. Pseudo-Ops and CMPSD**

Pseudo-Op	Implementation
CMPEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 0</i>
CMPLTSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 1</i>
CMPLES SD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 2</i>
CMPUNORD SD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 3</i>
CMPNEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 4</i>
CMPNLTS D <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 5</i>
CMPNLES D <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 6</i>
CMPORD SD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 7</i>

The greater-than relations not implemented in the processor 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.)

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

#### Enhanced Comparison Predicate for VEX-Encoded VCMPSD

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. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-14, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

**Table 3-14. Pseudo-Op and VCMPSD Implementation**

Pseudo-Op	CMPSD Implementation
VCMPEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0</i>
VCMPLESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1</i>
VCMPUNORDSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 2</i>
VCMPNEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 3</i>
VCMPNLTS D <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 4</i>
VCMPNLES D <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 5</i>
VCMPORDSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 6</i>
VCMPEQ_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 7</i>
VCMPNGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 8</i>
VCMPNGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 9</i>
VCMPFALSESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, OAH</i>
VCMPNEQ_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, OBH</i>
VCMPGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, OCH</i>
VCMPGTS D <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, ODH</i>
VCMPGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, OEH</i>

**Table 3-14. Pseudo-Op and VCMPSD Implementation (Contd.)**

Pseudo-Op	CMPSD Implementation
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
VCMPTURE_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1FH

## Operation

CASE (COMPARISON PREDICATE) OF

0: OP3  $\leftarrow$  EQ\_OQ; OP5  $\leftarrow$  EQ\_OQ;  
 1: OP3  $\leftarrow$  LT\_OS; OP5  $\leftarrow$  LT\_OS;  
 2: OP3  $\leftarrow$  LE\_OS; OP5  $\leftarrow$  LE\_OS;  
 3: OP3  $\leftarrow$  UNORD\_Q; OP5  $\leftarrow$  UNORD\_Q;  
 4: OP3  $\leftarrow$  NEQ\_UQ; OP5  $\leftarrow$  NEQ\_UQ;  
 5: OP3  $\leftarrow$  NLT\_US; OP5  $\leftarrow$  NLT\_US;  
 6: OP3  $\leftarrow$  NLE\_US; OP5  $\leftarrow$  NLE\_US;  
 7: OP3  $\leftarrow$  ORD\_Q; OP5  $\leftarrow$  ORD\_Q;  
 8: OP5  $\leftarrow$  EQ\_UQ;  
 9: OP5  $\leftarrow$  NGE\_US;  
 10: OP5  $\leftarrow$  NGT\_US;  
 11: OP5  $\leftarrow$  FALSE\_OQ;  
 12: OP5  $\leftarrow$  NEQ\_OQ;  
 13: OP5  $\leftarrow$  GE\_OS;  
 14: OP5  $\leftarrow$  GT\_OS;  
 15: OP5  $\leftarrow$  TRUE\_UQ;  
 16: OP5  $\leftarrow$  EQ\_OS;  
 17: OP5  $\leftarrow$  LT\_OQ;  
 18: OP5  $\leftarrow$  LE\_OQ;  
 19: OP5  $\leftarrow$  UNORD\_S;  
 20: OP5  $\leftarrow$  NEQ\_US;  
 21: OP5  $\leftarrow$  NLT\_UQ;  
 22: OP5  $\leftarrow$  NLE\_UQ;  
 23: OP5  $\leftarrow$  ORD\_S;  
 24: OP5  $\leftarrow$  EQ\_US;

```

25: OP5 ← 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;

```

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

```

CMPO ← DEST[63:0] OP3 SRC[63:0];
IF CMPO = TRUE
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0000000000000000H; FI;
DEST[VLMAX-1:64] (Unmodified)

```

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

```

CMPO ← SRC1[63:0] OP5 SRC2[63:0];
IF CMPO = TRUE
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0000000000000000H; FI;
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

CMPSD for equality: `_m128d _mm_cmpeq_sd(_m128d a, _m128d b)`  
 CMPSD for less-than: `_m128d _mm_cmplt_sd(_m128d a, _m128d b)`  
 CMPSD for less-than-or-equal: `_m128d _mm_cmple_sd(_m128d a, _m128d b)`  
 CMPSD for greater-than: `_m128d _mm_cmpgt_sd(_m128d a, _m128d b)`  
 CMPSD for greater-than-or-equal: `_m128d _mm_cmpge_sd(_m128d a, _m128d b)`  
 CMPSD for inequality: `_m128d _mm_cmpneq_sd(_m128d a, _m128d b)`  
 CMPSD for not-less-than: `_m128d _mm_cmplt_sd(_m128d a, _m128d b)`  
 CMPSD for not-greater-than: `_m128d _mm_cmplt_sd(_m128d a, _m128d b)`  
 CMPSD for not-greater-than-or-equal: `_m128d _mm_cmpnge_sd(_m128d a, _m128d b)`  
 CMPSD for ordered: `_m128d _mm_cmppord_sd(_m128d a, _m128d b)`  
 CMPSD for unordered: `_m128d _mm_cmppunord_sd(_m128d a, _m128d b)`  
 CMPSD for not-less-than-or-equal: `_m128d _mm_cmppnle_sd(_m128d a, _m128d b)`  
 VCMPSD: `_m128 _mm_cmp_sd(_m128 a, _m128 b, const int imm)`

**SIMD Floating-Point Exceptions**

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in above table, Denormal.

**Other Exceptions**

See Exceptions Type 3.

**CMPSS—Compare Scalar Single-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF C2 /r ib CMPSS <i>xmm1, xmm2/m32, imm8</i>	RMI	V/V	SSE	Compare low single-precision floating-point value in <i>xmm2/m32</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.LIG.F3.0F.WIG C2 /r ib VCMPSS <i>xmm1, xmm2, xmm3/m32, imm8</i>	RVMI	V/V	AVX	Compare low single precision floating-point value in <i>xmm3/m32</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as comparison predicate.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

Compares the low single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a double-word mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

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. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

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 a fault, since a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX =0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-15. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-15. Pseudo-Ops and CMPSS**

Pseudo-Op	CMPSS Implementation
CMPEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 0</i>
CMPLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 1</i>
CMPLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 2</i>
CMPUNORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 3</i>
CMPNEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 4</i>
CMPNLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 5</i>
CMPNLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 6</i>
CMPORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 7</i>

The greater-than relations not implemented in the processor 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.)

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

#### Enhanced Comparison Predicate for VEX-Encoded VCMPSD

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. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-9, 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. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

**Table 3-16. Pseudo-Op and VCMPSS Implementation**

Pseudo-Op	CMPSS Implementation
VCMPEQSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 0</i>
VCMPLTSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 1</i>
VCMPLESS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 2</i>
VCMPUNORDSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 3</i>
VCMPNEQSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 4</i>
VCMPNLTSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 5</i>
VCMPNLESS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 6</i>
VCMPORDSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 8</i>
VCMPNGESS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 9</i>
VCMPNGTSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSESS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 0CH</i>
VCMPGESS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 0DH</i>
VCMPGTSS <i>reg1, reg2, reg3</i>	VCMPSS <i>reg1, reg2, reg3, 0EH</i>

**Table 3-16. Pseudo-Op and VCMPSS Implementation (Contd.)**

Pseudo-Op	CMPSS Implementation
VCMPTRUSS 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
VCMPLLE_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

## Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3  $\leftarrow$  EQ\_OQ; OP5  $\leftarrow$  EQ\_OQ;
- 1: OP3  $\leftarrow$  LT\_OS; OP5  $\leftarrow$  LT\_OS;
- 2: OP3  $\leftarrow$  LE\_OS; OP5  $\leftarrow$  LE\_OS;
- 3: OP3  $\leftarrow$  UNORD\_Q; OP5  $\leftarrow$  UNORD\_Q;
- 4: OP3  $\leftarrow$  NEQ\_UQ; OP5  $\leftarrow$  NEQ\_UQ;
- 5: OP3  $\leftarrow$  NLT\_US; OP5  $\leftarrow$  NLT\_US;
- 6: OP3  $\leftarrow$  NLE\_US; OP5  $\leftarrow$  NLE\_US;
- 7: OP3  $\leftarrow$  ORD\_Q; OP5  $\leftarrow$  ORD\_Q;
- 8: OP5  $\leftarrow$  EQ\_UQ;
- 9: OP5  $\leftarrow$  NGE\_US;
- 10: OP5  $\leftarrow$  NGT\_US;
- 11: OP5  $\leftarrow$  FALSE\_OQ;
- 12: OP5  $\leftarrow$  NEQ\_OQ;
- 13: OP5  $\leftarrow$  GE\_OS;
- 14: OP5  $\leftarrow$  GT\_OS;
- 15: OP5  $\leftarrow$  TRUE\_UQ;
- 16: OP5  $\leftarrow$  EQ\_OS;
- 17: OP5  $\leftarrow$  LT\_OQ;
- 18: OP5  $\leftarrow$  LE\_OQ;
- 19: OP5  $\leftarrow$  UNORD\_S;
- 20: OP5  $\leftarrow$  NEQ\_US;
- 21: OP5  $\leftarrow$  NLT\_UQ;
- 22: OP5  $\leftarrow$  NLE\_UQ;
- 23: OP5  $\leftarrow$  ORD\_S;
- 24: OP5  $\leftarrow$  EQ\_US;

```

25: OP5 ← 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;

```

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

```

CMPO ← DEST[31:0] OP3 SRC[31:0];
IF CMPO = TRUE
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 00000000H; FI;
DEST[VLMAX-1:32] (Unmodified)

```

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

```

CMPO ← SRC1[31:0] OP5 SRC2[31:0];
IF CMPO = TRUE
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 00000000H; FI;
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

CMPSS for equality: `__m128_mm_cmpeq_ss(__m128 a, __m128 b)`  
 CMPSS for less-than: `__m128_mm_cmplt_ss(__m128 a, __m128 b)`  
 CMPSS for less-than-or-equal: `__m128_mm_cmple_ss(__m128 a, __m128 b)`  
 CMPSS for greater-than: `__m128_mm_cmpgt_ss(__m128 a, __m128 b)`  
 CMPSS for greater-than-or-equal: `__m128_mm_cmpge_ss(__m128 a, __m128 b)`  
 CMPSS for inequality: `__m128_mm_cmpeq_ss(__m128 a, __m128 b)`  
 CMPSS for not-less-than: `__m128_mm_cmplt_ss(__m128 a, __m128 b)`  
 CMPSS for not-greater-than: `__m128_mm_cmple_ss(__m128 a, __m128 b)`  
 CMPSS for not-greater-than-or-equal: `__m128_mm_cmpte_ss(__m128 a, __m128 b)`  
 CMPSS for ordered: `__m128_mm_cmport_ss(__m128 a, __m128 b)`  
 CMPSS for unordered: `__m128_mm_cmponord_ss(__m128 a, __m128 b)`  
 CMPSS for not-less-than-or-equal: `__m128_mm_cmpte_ss(__m128 a, __m128 b)`  
 VCMPPS: `__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 above table, Denormal.

**Other Exceptions**

See Exceptions Type 3.

## CMPXCHG—Compare and Exchange

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF B0/r CMPXCHG <i>r/m8, r8</i>	MR	Valid	Valid*	Compare AL with <i>r/m8</i> . If equal, ZF is set and <i>r8</i> is loaded into <i>r/m8</i> . Else, clear ZF and load <i>r/m8</i> into AL.
REX + OF B0/r CMPXCHG <i>r/m8**,r8</i>	MR	Valid	N.E.	Compare AL with <i>r/m8</i> . If equal, ZF is set and <i>r8</i> is loaded into <i>r/m8</i> . Else, clear ZF and load <i>r/m8</i> into AL.
OF B1/r CMPXCHG <i>r/m16, r16</i>	MR	Valid	Valid*	Compare AX with <i>r/m16</i> . If equal, ZF is set and <i>r16</i> is loaded into <i>r/m16</i> . Else, clear ZF and load <i>r/m16</i> into AX.
OF B1/r CMPXCHG <i>r/m32, r32</i>	MR	Valid	Valid*	Compare EAX with <i>r/m32</i> . If equal, ZF is set and <i>r32</i> is loaded into <i>r/m32</i> . Else, clear ZF and load <i>r/m32</i> into EAX.
REX.W + OF B1/r CMPXCHG <i>r/m64, r64</i>	MR	Valid	N.E.	Compare RAX with <i>r/m64</i> . If equal, ZF is set and <i>r64</i> is loaded into <i>r/m64</i> . Else, clear ZF and load <i>r/m64</i> into RAX.

### NOTES:

\* See the IA-32 Architecture Compatibility section below.

\*\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

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

### 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;
Fl;

```

### 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**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
OF C7 /1 m64	M	Valid	Valid*	Compare EDX:EAX with <i>m64</i> . If equal, set ZF and load ECX:EBX into <i>m64</i> . Else, clear ZF and load <i>m64</i> into EDX:EAX.
CMPXCHG8B <i>m64</i>				
REX.W + OF C7 /1 m128	M	Valid	N.E.	Compare RDX:RAX with <i>m128</i> . If equal, set ZF and load RCX:RBX into <i>m128</i> . Else, clear ZF and load <i>m128</i> into RDX:RAX.
CMPXCHG16B <i>m128</i>				

**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)	NA	NA	NA

**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 page 3-170.

**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;

```

### 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[bit 13] = 0.

#UD If the destination operand is not a memory location.

## INSTRUCTION SET REFERENCE, A-M

- #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	CPUID Feature Flag	Description
66 0F 2F /r COMISD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Compare low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD <i>xmm1, xmm2/m64</i>		RM V/V	AVX	Compare low double precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.

**Instruction Operand Encoding**

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

**Description**

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). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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**

```
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 Equivalents**

```
int _mm_comieq_sd (__m128d a, __m128d b)
int _mm_comilt_sd (__m128d a, __m128d b)
int _mm_comile_sd (__m128d a, __m128d b)
int _mm_comigt_sd (__m128d a, __m128d b)
int _mm_comige_sd (__m128d a, __m128d b)
int _mm_comineq_sd (__m128d a, __m128d b)
```

### SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

### Other Exceptions

See Exceptions Type 3; additionally

#UD                    If VEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 2F /r COMISS <i>xmm1</i> , <i>xmm2/m32</i>		RM V/V	SSE	Compare low single-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.
VEX.LIG.OF.WIG 2F /r VCOMISS <i>xmm1</i> , <i>xmm2/m32</i>		RM V/V	AVX	Compare low single precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.

### Instruction Operand Encoding

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

### Description

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). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

Operand 1 is an XMM register; Operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

```
RESULT ← OrderedCompare(SRC1[31:0] <> SRC2[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED:      ZF,PF,CF ← 111;
    GREATER_THAN:   ZF,PF,CF ← 000;
    LESS_THAN:      ZF,PF,CF ← 001;
    EQUAL:          ZF,PF,CF ← 100;
ESAC;
OF,AF,SF ← 0; }
```

### Intel C/C++ Compiler Intrinsic Equivalents

```
int _mm_comieq_ss (__m128 a, __m128 b)
int _mm_comilt_ss (__m128 a, __m128 b)
int _mm_comile_ss (__m128 a, __m128 b)
int _mm_comigt_ss (__m128 a, __m128 b)
int _mm_comige_ss (__m128 a, __m128 b)
int _mm_comineq_ss (__m128 a, __m128 b)
```

### SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

### Other Exceptions

See Exceptions Type 3; additionally

#UD                    If VEX.vvvv != 1111B.

## CPUID—CPU Identification

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

### Instruction Operand Encoding

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

### Description

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

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

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

Table 3-17 shows information returned, depending on the initial value loaded into the EAX register. Table 3-18 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

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

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers. For example, using the Intel Core i7 processor, the following is true:

```
CPUID.EAX = 07H (*Returns EAX=EBX=ECX=EDX=0. *)
```

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

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

### See also:

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

---

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

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

**Table 3-17. Information Returned by CPUID Instruction**

<b>Initial EAX Value</b>	<b>Information Provided about the Processor</b>					
<i>Basic CPUID Information</i>						
0H	EAX	Maximum Input Value for Basic CPUID Information (see Table 3-18)				
	EBX	"Genu"				
	ECX	"ntel"				
	EDX	"inel"				
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-5)				
	EBX	Bits 07-00: Brand Index Bits 15-08: CLFLUSH line size (Value * 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID				
	ECX	Feature Information (see Figure 3-6 and Table 3-20)				
	EDX	Feature Information (see Figure 3-7 and Table 3-21)				
	<b>NOTES:</b>					
	* The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.					
02H	EAX	Cache and TLB Information (see Table 3-22)				
	EBX	Cache and TLB Information				
	ECX	Cache and TLB Information				
	EDX	Cache and TLB Information				
03H	EAX	Reserved.				
	EBX	Reserved.				
	ECX	Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)				
	EDX	Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)				
	<b>NOTES:</b>					
	Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.					
	See AP-485, <i>Intel Processor Identification and the CPUID Instruction</i> (Order Number 241618) for more information on PSN.					
CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLE.BOOT_NT4[bit 22] = 0 (default).						
<i>Deterministic Cache Parameters Leaf</i>						
04H	<b>NOTES:</b>					
	Leaf 04H output depends on the initial value in ECX.*					
	See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 3-178.					
	EAX	Bits 04-00: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved				

**Table 3-17. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	<p>Bits 07-05: Cache Level (starts at 1)        Bit 08: Self Initializing cache level (does not need SW initialization)        Bit 09: Fully Associative cache        Bits 13-10: Reserved        Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache**, ***        Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****</p> <p>EBX      Bits 11-00: L = System Coherency Line Size**        Bits 21-12: P = Physical Line partitions**        Bits 31-22: W = Ways of associativity**</p> <p>ECX      Bits 31-00: S = Number of Sets**</p> <p>EDX      Bit 0: Write-Back Invalidate/Invalidate        0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache.        1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.        Bit 1: Cache Inclusiveness        0 = Cache is not inclusive of lower cache levels.        1 = Cache is inclusive of lower cache levels.        Bit 2: Complex Cache Indexing        0 = Direct mapped cache.        1 = A complex function is used to index the cache, potentially using all address bits.        Bits 31-03: Reserved = 0</p> <p><b>NOTES:</b></p> <ul style="list-style-type: none"> <li>* If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Invalid sub-leaves of EAX = 04H: ECX = n, n &gt; 3.</li> <li>** Add one to the return value to get the result.</li> <li>***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache</li> <li>**** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID.</li> <li>***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</li> </ul>	
<i>MONITOR/MwAIT Leaf</i>		
05H	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	ECX	Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled Bits 31 - 02: Reserved

**Table 3-17. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWAIT Bits 07 - 04: Number of C1* sub C-states supported using MWAIT Bits 11 - 08: Number of C2* sub C-states supported using MWAIT Bits 15 - 12: Number of C3* sub C-states supported using MWAIT Bits 19 - 16: Number of C4* sub C-states supported using MWAIT Bits 23 - 20: Number of C5* sub C-states supported using MWAIT Bits 27 - 24: Number of C6* sub C-states supported using MWAIT Bits 31 - 28: Number of C7* sub C-states supported using MWAIT <b>NOTE:</b> * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
<i>Thermal and Power Management Leaf</i>		
06H	EAX	Bit 00: Digital temperature sensor is supported if set Bit 01: Intel Turbo Boost Technology Available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENALBE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bits 31 - 15: Reserved
<i>Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)</i>		
07H	EAX	Sub-leaf 0 (Input ECX = 0). *  Bits 31-00: Reports the maximum input value for supported leaf 7 sub-leaves.

**Table 3-17. Information Returned by CPUID Instruction (Contd.)**

<b>Initial EAX Value</b>	<b>Information Provided about the Processor</b>	
	EBX	<p>Bit 00: FSGSBASE. Supports RDGSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1.</p> <p>Bit 01: IA32_TSC_ADJUST MSR is supported if 1.</p> <p>Bit 02: Reserved</p> <p>Bit 03: BMI1</p> <p>Bit 04: HLE</p> <p>Bit 05: AVX2</p> <p>Bit 06: Reserved</p> <p>Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1.</p> <p>Bit 08: BMI2</p> <p>Bit 09: Supports Enhanced REP MOVSB/STOSB if 1.</p> <p>Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers.</p> <p>Bit 11: RTM</p> <p>Bit 12: Supports Platform Quality of Service Monitoring (PQM) capability if 1.</p> <p>Bit 13: Deprecates FPU CS and FPU DS values if 1.</p> <p>Bit 14: Reserved.</p> <p>Bit 15: Supports Platform Quality of Service Enforcement (PQE) capability if 1.</p> <p>Bits 31:16: Reserved</p>
<b>NOTE:</b>		
	ECX	Reserved
	EDX	Reserved
<i>Direct Cache Access Information Leaf</i>		
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H)
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
<i>Architectural Performance Monitoring Leaf</i>		
0AH	EAX	<p>Bits 07 - 00: Version ID of architectural performance monitoring</p> <p>Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor</p> <p>Bits 23 - 16: Bit width of general-purpose, performance monitoring counter</p> <p>Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events</p>
	EBX	<p>Bit 00: Core cycle event not available if 1</p> <p>Bit 01: Instruction retired event not available if 1</p> <p>Bit 02: Reference cycles event not available if 1</p> <p>Bit 03: Last-level cache reference event not available if 1</p> <p>Bit 04: Last-level cache misses event not available if 1</p> <p>Bit 05: Branch instruction retired event not available if 1</p> <p>Bit 06: Branch mispredict retired event not available if 1</p> <p>Bits 31 - 07: Reserved = 0</p>
	ECX	Reserved = 0
	EDX	<p>Bits 04 - 00: Number of fixed-function performance counters (if Version ID &gt; 1)</p> <p>Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID &gt; 1)</p> <p>Reserved = 0</p>
<i>Extended Topology Enumeration Leaf</i>		

**Table 3-17. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
OBH	<p><b>NOTES:</b></p> <p>Most of Leaf OBH output depends on the initial value in ECX.</p> <p>The EDX output of leaf OBH is always valid and does not vary with input value in ECX.</p> <p>Output value in ECX[7:0] always equals input value in ECX[7:0].</p> <p>For sub-leaves that return an invalid level-type of 0 in ECX[15:8], EAX and EBX will return 0.</p> <p>If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX &gt; n also return 0 in ECX[15:8].</p>
<i>Processor Extended State Enumeration Main Leaf (EAX = ODH, ECX = 0)</i>	
ODH	<p><b>NOTES:</b></p> <p>Leaf ODH main leaf (ECX = 0).</p>
EAX	<p>Bits 31-00: Reports the valid bit fields of the lower 32 bits of XCRO. If a bit is 0, the corresponding bit field in XCRO is reserved.</p> <p>Bit 00: legacy x87</p> <p>Bit 01: 128-bit SSE</p> <p>Bit 02: 256-bit AVX</p> <p>Bits 31- 03: Reserved</p>
EBX	<p>Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCRO. May be different than ECX if some features at the end of the XSAVE save area are not enabled.</p>
ECX	<p>Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCRO.</p>
EDX	<p>Bit 31-00: Reports the valid bit fields of the upper 32 bits of XCRO. If a bit is 0, the corresponding bit field in XCRO is reserved.</p>

**Table 3-17. Information Returned by CPUID Instruction (Contd.)**

<b>Initial EAX Value</b>	<b>Information Provided about the Processor</b>	
<i>Processor Extended State Enumeration Sub-leaf (EAX = ODH, ECX = 1)</i>		
ODH	EAX	<p>Bits 31-04: Reserved</p> <p>Bit 00: XSAVEOPT is available</p> <p>Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set</p> <p>Bit 02: Supports XGETBV with ECX = 1 if set</p> <p>Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set</p>
<i>Processor Extended State Enumeration Sub-leaves (EAX = ODH, ECX = n, n &gt; 1)</i>		
ODH		<p><b>NOTES:</b></p> <p>Leaf ODH output depends on the initial value in ECX.</p> <p>Each valid sub-leaf index maps to a valid bit in either the XCRO register or the IA32_XSS MSR starting at bit position 2.</p> <p>* If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Invalid sub-leaves of EAX = ODH: ECX = n, n &gt; 2.</p>
	EAX	<p>Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, n. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCRO register*.</p>
	EBX	<p>Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area.</p> <p>This field reports 0 if the sub-leaf index, n, is invalid*.</p>
	ECX	<p>This field reports 0 if the sub-leaf index, n, is invalid*; otherwise, bit 0 is set if the sub-leaf index, n, maps to a valid bit in the IA32_XSS MSR, and bits 31-1 are reserved.</p>
	EDX	<p>This field reports 0 if the sub-leaf index, n, is invalid*; otherwise it is reserved.</p>
<i>Platform QoS Monitoring Enumeration Sub-leaf (EAX = OFH, ECX = 0)</i>		
OFH		<p><b>NOTES:</b></p> <p>Leaf OFH output depends on the initial value in ECX.</p> <p>Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX</p>
	EAX	Reserved.
	EBX	Bits 31-0: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	<p>Bit 00: Reserved.</p> <p>Bit 01: Supports L3 Cache QoS Monitoring if 1.</p> <p>Bits 31:02: Reserved</p>
<i>L3 Cache QoS Monitoring Capability Enumeration Sub-leaf (EAX = OFH, ECX = 1)</i>		

**Table 3-17. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor		
OFH	<b>NOTES:</b> Leaf OFH output depends on the initial value in ECX.		
	EAX	Reserved.	
	EBX	Bits 31:0: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).	
	ECX	Maximum range (zero-based) of RMID of this resource type.	
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bits 31:01: Reserved	
<i>Platform QoS Enforcement Enumeration Sub-leaf (EAX = 10H, ECX = 0)</i>			
10H	<b>NOTES:</b> Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EDX		
	EAX	Reserved.	
	EBX	Bit 00: Reserved. Bit 01: Supports L3 Cache QoS Enforcement if 1. Bits 31:02: Reserved	
	ECX	Reserved.	
	EDX	Reserved.	
<i>L3 Cache QoS Enforcement Enumeration Sub-leaf (EAX = 10H, ECX = ResID =1)</i>			
10H	<b>NOTES:</b> Leaf 10H output depends on the initial value in ECX.		
	EAX	Bits 4:0: Length of the capacity bit mask for the corresponding ResID. Bits 31:05: Reserved	
	EBX	Bits 31:0: Bit-granular map of isolation/contention of allocation units.	
	ECX	Bit 00: Reserved. Bit 01: Updates of COS should be infrequent if 1. Bits 31:02: Reserved	
	EDX	Bits 15:0: Highest COS number supported for this ResID. Bits 31:16: Reserved	
<i>Unimplemented CPUID Leaf Functions</i>			
40000000H - 4FFFFFFFH	Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.		
<i>Extended Function CPUID Information</i>			
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 3-18).	
	EBX	Reserved	
	ECX	Reserved	
	EDX	Reserved	

**Table 3-17. Information Returned by CPUID Instruction (Contd.)**

<b>Initial EAX Value</b>	<b>Information Provided about the Processor</b>		
80000001H	EAX	Extended Processor Signature and Feature Bits.	
	EBX	Reserved	
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode Bits 04-01 Reserved Bit 05: LZCNT Bits 07-06 Reserved Bit 08: PREFETCHW Bits 31-09 Reserved	
	EDX	Bits 10-00: Reserved Bit 11: SYSCALL/SYSRET available in 64-bit mode Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 25-21: Reserved = 0 Bit 26: 1-GByte pages are available if 1 Bit 27: RDTSCP and IA32_TSC_AUX are available if 1 Bits 28: Reserved = 0 Bit 29: Intel® 64 Architecture available if 1 Bits 31-30: Reserved = 0	
80000002H	EAX	Processor Brand String	
	EBX	Processor Brand String Continued	
	ECX	Processor Brand String Continued	
	EDX	Processor Brand String Continued	
80000003H	EAX	Processor Brand String Continued	
	EBX	Processor Brand String Continued	
	ECX	Processor Brand String Continued	
	EDX	Processor Brand String Continued	
80000004H	EAX	Processor Brand String Continued	
	EBX	Processor Brand String Continued	
	ECX	Processor Brand String Continued	
	EDX	Processor Brand String Continued	
80000005H	EAX	Reserved = 0	
	EBX	Reserved = 0	
	ECX	Reserved = 0	
	EDX	Reserved = 0	
80000006H	EAX	Reserved = 0	
	EBX	Reserved = 0	
	ECX	Bits 07-00: Cache Line size in bytes Bits 11-08: Reserved Bits 15-12: L2 Associativity field *	
	EDX	Bits 31-16: Cache size in 1K units Reserved = 0	

**Table 3-17. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor		
	<b>NOTES:</b> * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative		
80000007H	EAX	Reserved = 0	
	EBX	Reserved = 0	
	ECX	Reserved = 0	
	EDX	Bits 07-00: Reserved = 0 Bit 08: Invariant TSC available if 1 Bits 31-09: Reserved = 0	
80000008H	EAX	Linear/Physical Address size Bits 07-00: #Physical Address Bits* Bits 15-8: #Linear Address Bits Bits 31-16: Reserved = 0	
	EBX	Reserved = 0	
	ECX	Reserved = 0	
	EDX	Reserved = 0	
	<b>NOTES:</b> * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.		

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

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 3-18) and is processor specific.

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

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

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

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

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

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 3-18) and is processor specific.

**Table 3-18. Highest CPUID Source Operand for Intel 64 and IA-32 Processors**

Intel 64 or IA-32 Processors	Highest Value in EAX	
	Basic Information	Extended Function Information
Earlier Intel486 Processors	CPUID Not Implemented	CPUID Not Implemented
Later Intel486 Processors and Pentium Processors	01H	Not Implemented

**Table 3-18. Highest CPUID Source Operand for Intel 64 and IA-32 Processors (Contd.)**

Intel 64 or IA-32 Processors	Highest Value in EAX	
	Basic Information	Extended Function Information
Pentium Pro and Pentium II Processors, Intel® Celeron® Processors	02H	Not Implemented
Pentium III Processors	03H	Not Implemented
Pentium 4 Processors	02H	80000004H
Intel Xeon Processors	02H	80000004H
Pentium M Processor	02H	80000004H
Pentium 4 Processor supporting Hyper-Threading Technology	05H	80000008H
Pentium D Processor (8xx)	05H	80000008H
Pentium D Processor (9xx)	06H	80000008H
Intel Core Duo Processor	0AH	80000008H
Intel Core 2 Duo Processor	0AH	80000008H
Intel Xeon Processor 3000, 5100, 5200, 5300, 5400 Series	0AH	80000008H
Intel Core 2 Duo Processor 8000 Series	0DH	80000008H
Intel Xeon Processor 5200, 5400 Series	0AH	80000008H
Intel Atom Processor	0AH	80000008H
Intel Core i7 Processor	0BH	80000008H

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

For processors that support the microcode update facility, the IA32\_BIOS\_SIGN\_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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

When CPUID executes with EAX set to 1, version information is returned in EAX (see Figure 3-5). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

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

See Table 3-19 for available processor type values. Stepping IDs are provided as needed.

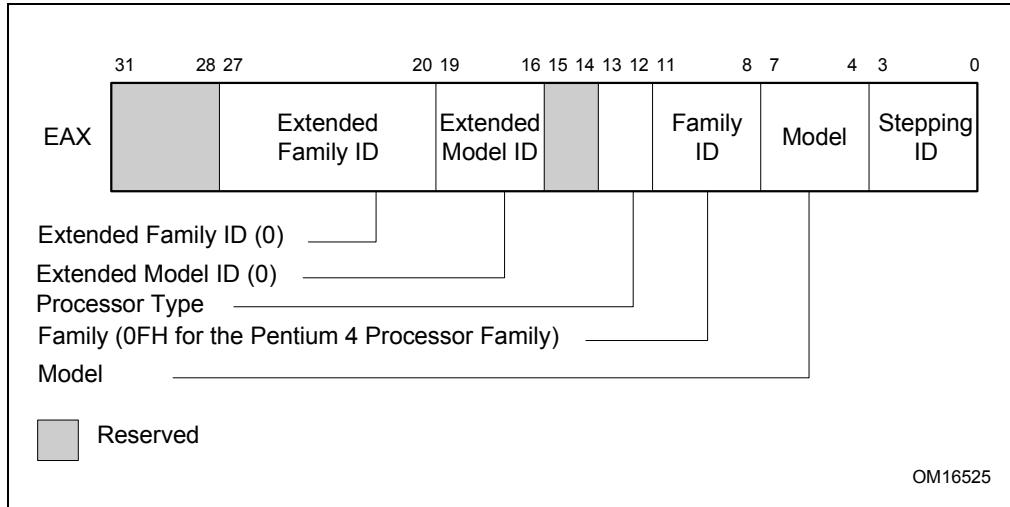


Figure 3-5. Version Information Returned by CPUID in EAX

**Table 3-19. Processor Type Field**

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

**NOTE**

See Chapter 17 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for information on identifying earlier IA-32 processors.

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

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

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

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

**INPUT EAX = 1: Returns Additional Information in EBX**

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

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

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

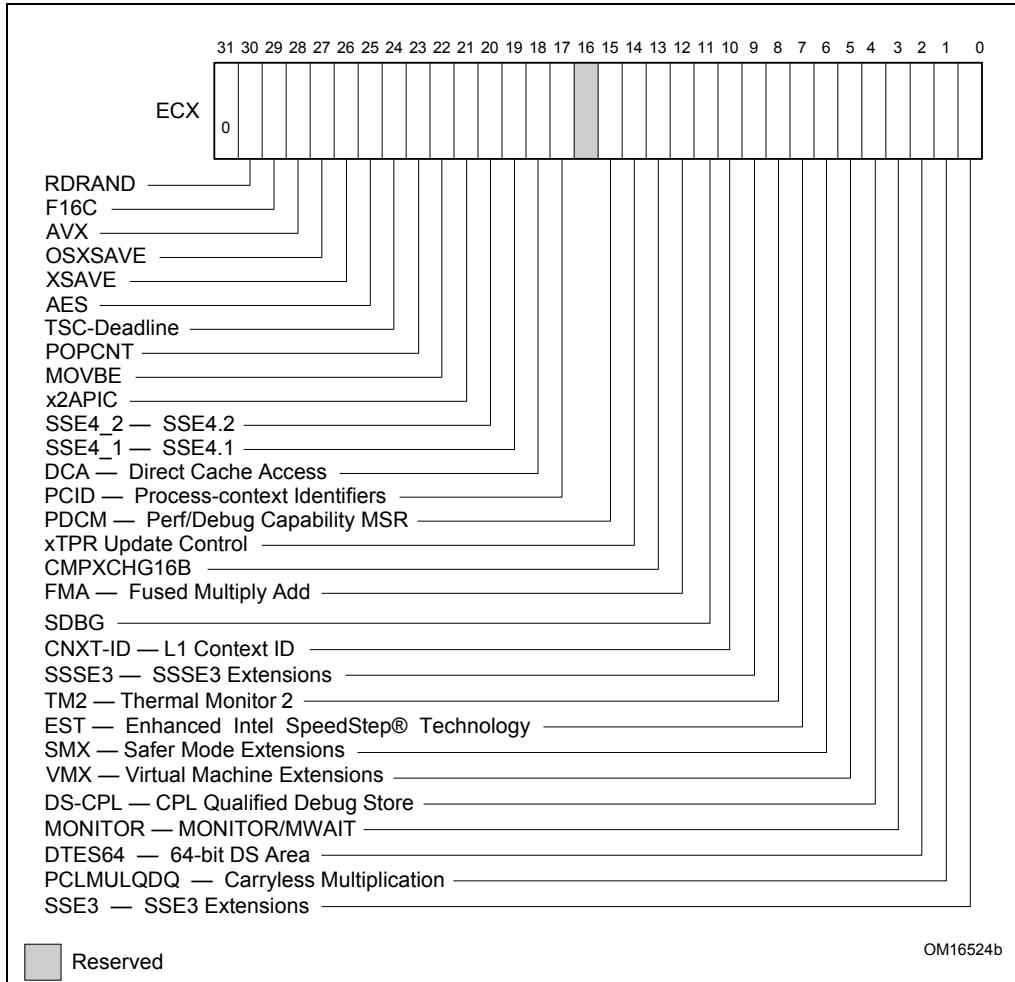
When CPUID executes with EAX set to 1, feature information is returned in ECX and EDX.

- Figure 3-6 and Table 3-20 show encodings for ECX.
- Figure 3-7 and Table 3-21 show encodings for EDX.

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

**NOTE**

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

**Figure 3-6. Feature Information Returned in the ECX Register****Table 3-20. Feature Information Returned in the ECX Register**

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

**Table 3-20. Feature Information Returned in the ECX Register (Contd.)**

Bit #	Mnemonic	Description
10	CNXT-ID	<b>L1 Context ID.</b> A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	<b>CMPXCHG16B Available.</b> A value of 1 indicates that the feature is available. See the "CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes" section in this chapter for a description.
14	xTPR Update Control	<b>xTPR Update Control.</b> A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	<b>Perfmon and Debug Capability:</b> A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	<b>Process-context identifiers.</b> A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCRO.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable the XSAVE feature set.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.

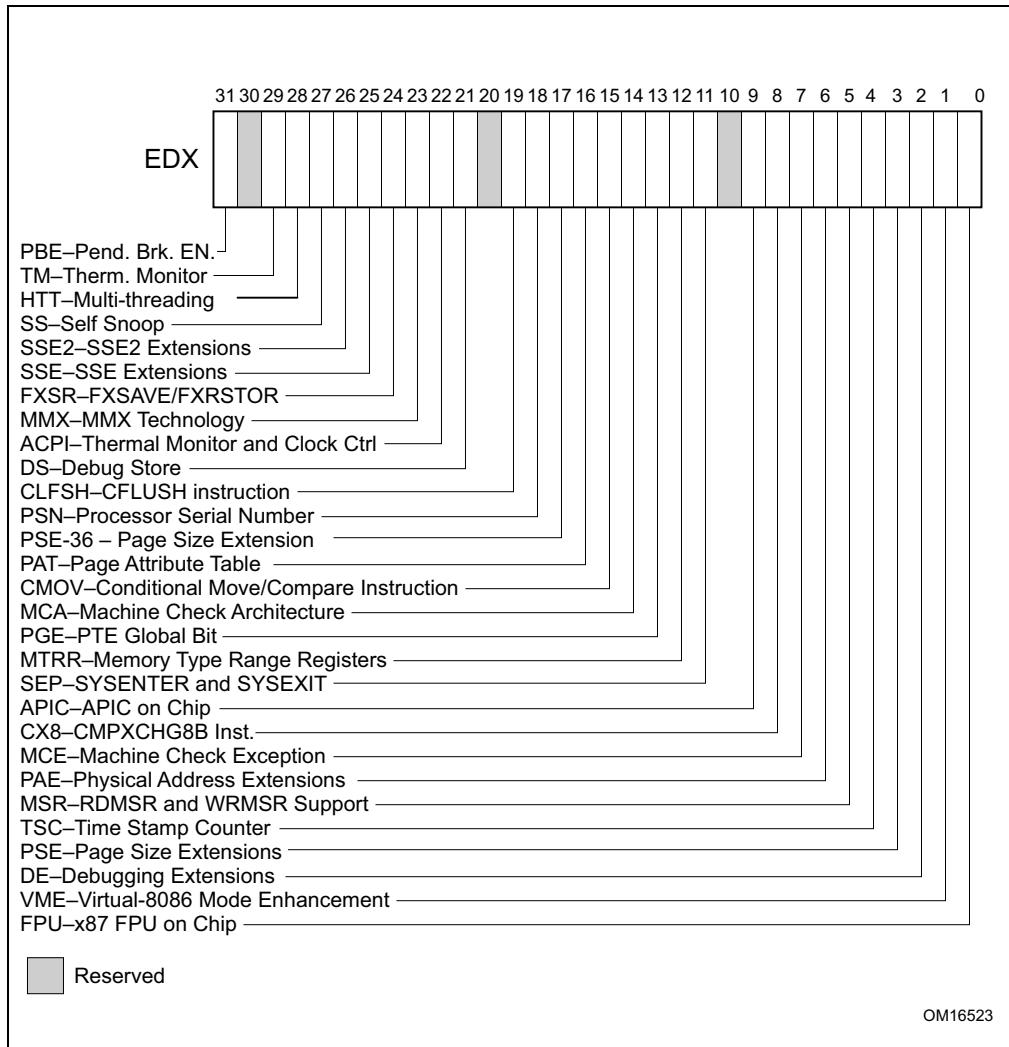


Figure 3-7. Feature Information Returned in the EDX Register

**Table 3-21. More on Feature Information Returned in the EDX Register**

Bit #	Mnemonic	Description
0	FPU	<b>Floating Point Unit On-Chip.</b> The processor contains an x87 FPU.
1	VME	<b>Virtual 8086 Mode Enhancements.</b> Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	<b>Debugging Extensions.</b> Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	<b>Page Size Extension.</b> Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	<b>Time Stamp Counter.</b> The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	<b>Model Specific Registers RDMSR and WRMSR Instructions.</b> The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	<b>Physical Address Extension.</b> Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	<b>Machine Check Exception.</b> Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	<b>CMPXCHG8B Instruction.</b> The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	<b>APIC On-Chip.</b> The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	<b>SYSENTER and SYSEXIT Instructions.</b> The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	<b>Memory Type Range Registers.</b> MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	<b>Page Global Bit.</b> The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	<b>Machine Check Architecture.</b> The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	<b>Conditional Move Instructions.</b> The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	<b>Page Attribute Table.</b> Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	<b>36-Bit Page Size Extension.</b> 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	<b>Processor Serial Number.</b> The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	<b>CLFLUSH Instruction.</b> CLFLUSH Instruction is supported.
20	Reserved	Reserved

**Table 3-21. More on Feature Information Returned in the EDX Register (Contd.)**

Bit #	Mnemonic	Description
21	DS	<b>Debug Store.</b> The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 23, “Introduction to Virtual-Machine Extensions,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C</i> ).
22	ACPI	<b>Thermal Monitor and Software Controlled Clock Facilities.</b> The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	<b>Intel MMX Technology.</b> The processor supports the Intel MMX technology.
24	FXSR	<b>FXSAVE and FXRSTOR Instructions.</b> The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	<b>SSE.</b> The processor supports the SSE extensions.
26	SSE2	<b>SSE2.</b> The processor supports the SSE2 extensions.
27	SS	<b>Self Snoop.</b> The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	<b>Max APIC IDs reserved field is Valid.</b> A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	<b>Thermal Monitor.</b> The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	<b>Pending Break Enable.</b> The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

**INPUT EAX = 2: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX**

When CPUID executes with EAX set to 2, the processor returns information about the processor’s internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor’s TLB/Cache/Prefetch hardware. The Intel Xeon processor 7400 series will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-22. Table 3-22 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of “cache type” via CPUID leaf 2.

**Table 3-22. Encoding of CPUID Leaf 2 Descriptors**

<b>Value</b>	<b>Type</b>	<b>Description</b>
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

**Table 3-22. Encoding of CPUID Leaf 2 Descriptors (Contd.)**

<b>Value</b>	<b>Type</b>	<b>Description</b>
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLBO: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLBO: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLBO: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLBO: 2-MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages, 128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages, 256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 1 GByte pages, 4-way set associative, 4 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Cache	Trace cache: 12 K-μop, 8-way set associative
71H	Cache	Trace cache: 16 K-μop, 8-way set associative
72H	Cache	Trace cache: 32 K-μop, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries

**Table 3-22. Encoding of CPUID Leaf 2 Descriptors (Contd.)**

<b>Value</b>	<b>Type</b>	<b>Description</b>
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
COH	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters

**Example 3-1. Example of Cache and TLB Interpretation**

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

```

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

```

Which means:

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

- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K-μop, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

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

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

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-17.

This Cache Size in Bytes

$$\begin{aligned} &= (\text{Ways} + 1) * (\text{Partitions} + 1) * (\text{Line\_Size} + 1) * (\text{Sets} + 1) \\ &= (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1) \end{aligned}$$

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-17.

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

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-17.

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

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-17.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-17), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

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

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-17.

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

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-17) is greater than Pn 0. See Table 3-17.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, "Introduction to Virtual-Machine Extensions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

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

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is  $\geq$  0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-17.

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

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-17.

When CPUID executes with EAX set to 0DH and ECX = n ( $n > 1$ , and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-17. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
  IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1 ) // VECTOR is the 64-bit value of EDX:EAX
    Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
  Fl;
```

## INPUT EAX = 0FH: Returns Platform Quality of Service (PQoS) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-17.

When CPUID executes with EAX set to 0FH and ECX = n ( $n \geq 1$ , and is a valid ResID), the processor returns information software can use to program IA32\_PQR\_ASSOC, IA32\_QM\_EVTSEL MSRs before reading QoS data from the IA32\_QM\_CTR MSR.

## INPUT EAX = 10H: Returns Platform Quality of Service (PQoS) Enforcement Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-17.

When CPUID executes with EAX set to 10H and ECX = n ( $n \geq 1$ , and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32\_resourceType\_Mask\_n.

## METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

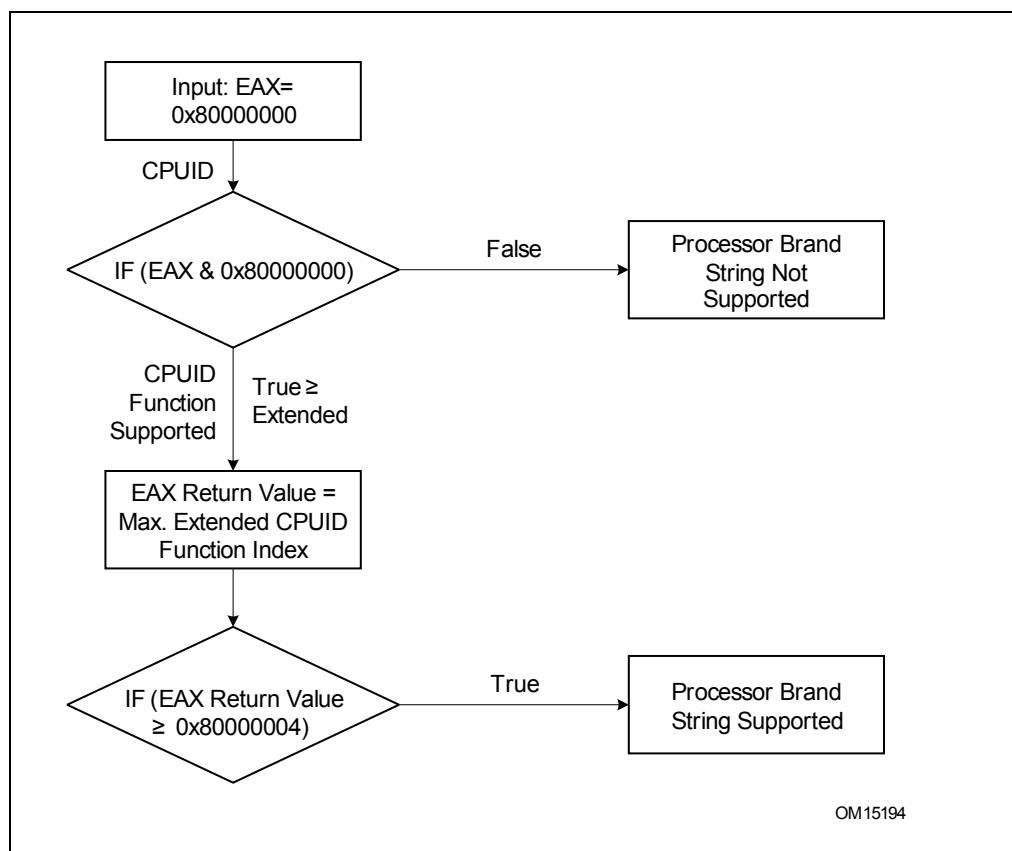
1. Processor brand string method; this method also returns the processor's maximum operating frequency
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## The Processor Brand String Method

Figure 3-8 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

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



**Figure 3-8. Determination of Support for the Processor Brand String**

## How Brand Strings Work

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

Table 3-23 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

**Table 3-23. Processor Brand String Returned with Pentium 4 Processor**

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

**Table 3-23. Processor Brand String Returned with Pentium 4 Processor (Contd.)**

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

### Extracting the Maximum Processor Frequency from Brand Strings

Figure 3-9 provides an algorithm which software can use to extract the maximum processor operating frequency from the processor brand string.

#### NOTE

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

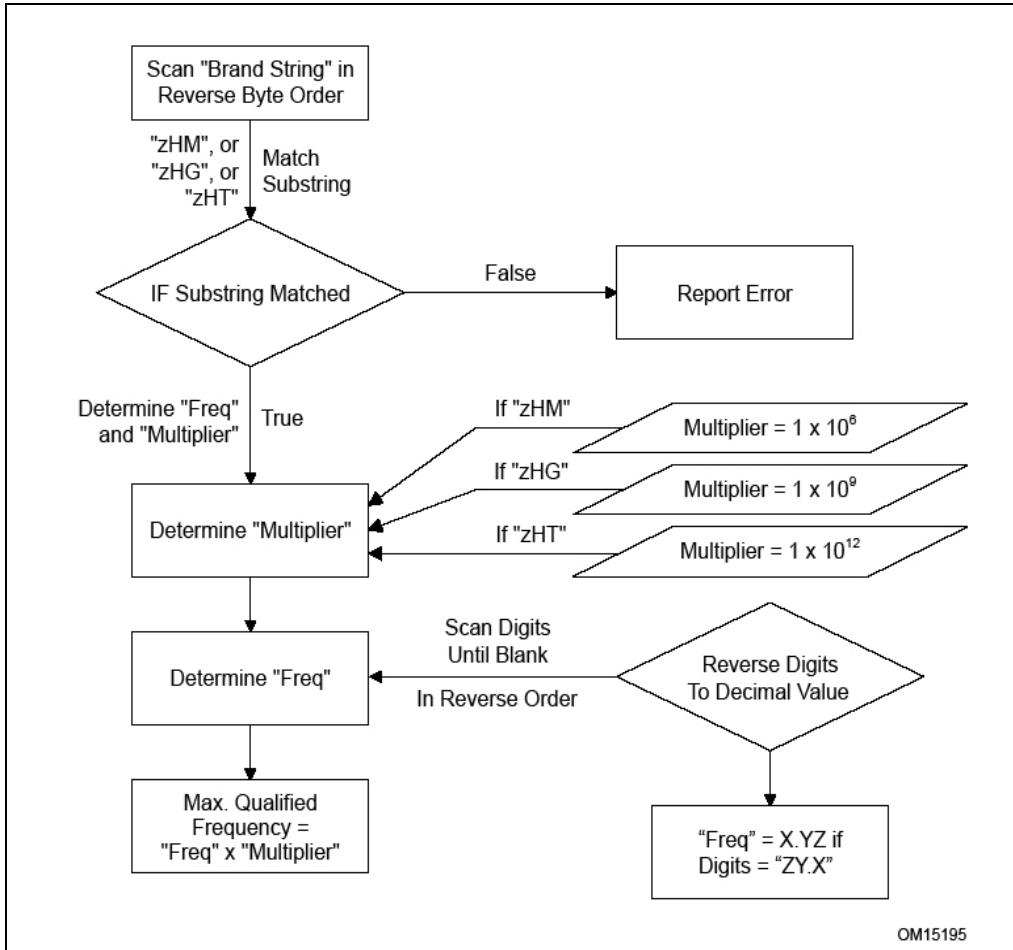


Figure 3-9. Algorithm for Extracting Maximum Processor Frequency

### The Processor Brand Index Method

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

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-24 shows brand indices that have identification strings associated with them.

Table 3-24. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor <sup>1</sup>
02H	Intel(R) Pentium(R) III processor <sup>1</sup>

**Table 3-24. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings**

03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor <sup>1</sup>
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
18H - OFFH	RESERVED

## NOTES:

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

**IA-32 Architecture Compatibility**

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

**Operation**

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

CASE (EAX) OF

  EAX = 0:

    EAX ← Highest basic function input value understood by CPUID;  
     EBX ← Vendor identification string;  
     EDX ← Vendor identification string;  
     ECX ← Vendor identification string;

  BREAK;

  EAX = 1H:

    EAX[3:0] ← Stepping ID;  
     EAX[7:4] ← Model;  
     EAX[11:8] ← Family;  
     EAX[13:12] ← Processor type;  
     EAX[15:14] ← Reserved;  
     EAX[19:16] ← Extended Model;  
     EAX[27:20] ← Extended Family;

```

EAX[31:28] ← Reserved;
EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
EBX[15:8] ← CLFLUSH Line Size;
EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
EBX[24:31] ← Initial APIC ID;
ECX ← Feature flags; (* See Figure 3-6. *)
EDX ← Feature flags; (* See Figure 3-7. *)
BREAK;
EAX = 2H:
    EAX ← Cache and TLB information;
    EBX ← Cache and TLB information;
    ECX ← Cache and TLB information;
    EDX ← Cache and TLB information;
BREAK;
EAX = 3H:
    EAX ← Reserved;
    EBX ← Reserved;
    ECX ← ProcessorSerialNumber[31:0];
    (* Pentium III processors only, otherwise reserved. *)
    EDX ← ProcessorSerialNumber[63:32];
    (* Pentium III processors only, otherwise reserved. *)
BREAK
EAX = 4H:
    EAX ← Deterministic Cache Parameters Leaf; (* See Table 3-17. *)
    EBX ← Deterministic Cache Parameters Leaf;
    ECX ← Deterministic Cache Parameters Leaf;
    EDX ← Deterministic Cache Parameters Leaf;
BREAK;
EAX = 5H:
    EAX ← MONITOR/MWAIT Leaf; (* See Table 3-17. *)
    EBX ← MONITOR/MWAIT Leaf;
    ECX ← MONITOR/MWAIT Leaf;
    EDX ← MONITOR/MWAIT Leaf;
BREAK;
EAX = 6H:
    EAX ← Thermal and Power Management Leaf; (* See Table 3-17. *)
    EBX ← Thermal and Power Management Leaf;
    ECX ← Thermal and Power Management Leaf;
    EDX ← Thermal and Power Management Leaf;
BREAK;
EAX = 7H:
    EAX ← Structured Extended Feature Flags Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Structured Extended Feature Flags Enumeration Leaf;
    ECX ← Structured Extended Feature Flags Enumeration Leaf;
    EDX ← Structured Extended Feature Flags Enumeration Leaf;
BREAK;
EAX = 8H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 9H:
    EAX ← Direct Cache Access Information Leaf; (* See Table 3-17. *)

```

```

EBX ← Direct Cache Access Information Leaf;
ECX ← Direct Cache Access Information Leaf;
EDX ← Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
    EAX ← Architectural Performance Monitoring Leaf; (* See Table 3-17. *)
    EBX ← Architectural Performance Monitoring Leaf;
    ECX ← Architectural Performance Monitoring Leaf;
    EDX ← Architectural Performance Monitoring Leaf;
    BREAK
EAX = BH:
    EAX ← Extended Topology Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Extended Topology Enumeration Leaf;
    ECX ← Extended Topology Enumeration Leaf;
    EDX ← Extended Topology Enumeration Leaf;
BREAK;
EAX = CH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = DH:
    EAX ← Processor Extended State Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Processor Extended State Enumeration Leaf;
    ECX ← Processor Extended State Enumeration Leaf;
    EDX ← Processor Extended State Enumeration Leaf;
BREAK;
EAX = EH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = FH:
    EAX ← Platform Quality of Service Monitoring Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Platform Quality of Service Monitoring Enumeration Leaf;
    ECX ← Platform Quality of Service Monitoring Enumeration Leaf;
    EDX ← Platform Quality of Service Monitoring Enumeration Leaf;
BREAK;
EAX = 10H:
    EAX ← Platform Quality of Service Enforcement Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Platform Quality of Service Enforcement Enumeration Leaf;
    ECX ← Platform Quality of Service Enforcement Enumeration Leaf;
    EDX ← Platform Quality of Service Enforcement Enumeration Leaf;
BREAK;
EAX = 80000000H:
    EAX ← Highest extended function input value understood by CPUID;
    EBX ← Reserved;
    ECX ← Reserved;
    EDX ← Reserved;
BREAK;
EAX = 80000001H:

```

```

EAX ← Reserved;
EBX ← Reserved;
ECX ← Extended Feature Bits (* See Table 3-17.*);
EDX ← Extended Feature Bits (* See Table 3-17. *);
BREAK;
EAX = 80000002H:
    EAX ← Processor Brand String;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000003H:
    EAX ← Processor Brand String, continued;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000004H:
    EAX ← Processor Brand String, continued;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000005H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 80000006H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Cache information;
    EDX ← Reserved = 0;
BREAK;
EAX = 80000007H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = Misc Feature Flags;
BREAK;
EAX = 80000008H:
    EAX ← Reserved = Physical Address Size Information;
    EBX ← Reserved = Virtual Address Size Information;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX >= 40000000H and EAX <= 4FFFFFFFH:
DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
    (* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
    EAX ← Reserved; (* Information returned for highest basic information leaf. *)
    EBX ← Reserved; (* Information returned for highest basic information leaf. *)
    ECX ← Reserved; (* Information returned for highest basic information leaf. *)
    EDX ← Reserved; (* Information returned for highest basic information leaf. *)

```

BREAK;  
ESAC;

### Flags Affected

None.

### Exceptions (All Operating Modes)

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

**CRC32 – Accumulate CRC32 Value**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
F2 OF 38 F0 /r CRC32 r32, r/m8	RM	Valid	Valid	Accumulate CRC32 on r/m8.
F2 REX OF 38 F0 /r CRC32 r32, r/m8*	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 OF 38 F1 /r CRC32 r32, r/m16	RM	Valid	Valid	Accumulate CRC32 on r/m16.
F2 OF 38 F1 /r CRC32 r32, r/m32	RM	Valid	Valid	Accumulate CRC32 on r/m32.
F2 REX.W OF 38 F0 /r CRC32 r64, r/m8	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 REX.W OF 38 F1 /r CRC32 r64, r/m64	RM	Valid	N.E.	Accumulate CRC32 on r/m64.

**NOTES:**

\*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

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

**Description**

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 0x11EDC6F41) 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 [Bit 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 [Bit 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 [Bit 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 [Bit 20] = 0. If LOCK prefix is used.

## CVTQ2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF E6 CVTDQ2PD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE2	Convert two packed signed doubleword integers from <i>xmm2/m128</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.128.F3.0F.WIG E6 /r VCVTQ2PD <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Convert two packed signed doubleword integers from <i>xmm2/mem</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.256.F3.0F.WIG E6 /r VCVTQ2PD <i>ymm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed signed doubleword integers from <i>xmm2/mem</i> to four packed double-precision floating-point values in <i>ymm1</i> .

### Instruction Operand Encoding

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

### Description

Converts 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).

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

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding XMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

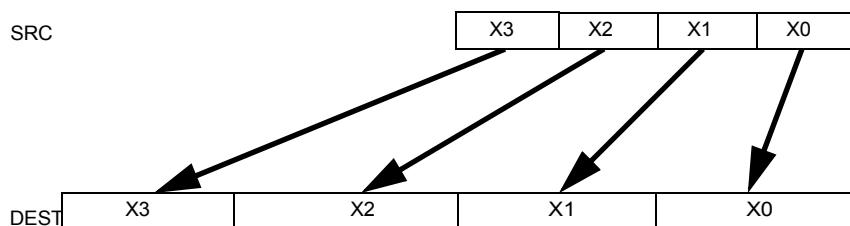


Figure 3-10. CVTDQ2PD (VEX.256 encoded version)

## Operation

### **CVTDQ2PD (128-bit Legacy SSE version)**

DEST[63:0]  $\leftarrow$  Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[127:64]  $\leftarrow$  Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[VLMAX-1:128] (unmodified)

### **VCVTDQ2PD (VEX.128 encoded version)**

DEST[63:0]  $\leftarrow$  Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[127:64]  $\leftarrow$  Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VCVTDQ2PD (VEX.256 encoded version)**

DEST[63:0]  $\leftarrow$  Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[127:64]  $\leftarrow$  Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[191:128]  $\leftarrow$  Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[255:192]  $\leftarrow$  Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[127:96])

## Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PD:    \_\_m128d \_mm\_cvtepi32\_pd(\_\_m128i a)  
 VCVTDQ2PD:   \_\_m256d \_mm256\_cvtepi32\_pd (\_\_m128i src)

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 5; additionally

#UD                      If VEX.vvvv != 1111B.

**CVTDQ2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values**

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

**Instruction Operand Encoding**

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

**Description**

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding XMM register destination are unmodified.

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

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

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

**Operation****CVTDQ2PS (128-bit Legacy SSE version)**

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[VLMAX-1:128] (unmodified)
```

**VCVTDQ2PS (VEX.128 encoded version)**

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[VLMAX-1:128] ← 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[127z: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])

```

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

CVTDQ2PS: `__m128 _mm_cvtepi32_ps(__m128i a)`  
 VCVTDQ2PS: `__m256 _mm256_cvtepi32_ps (__m256i src)`

**SIMD Floating-Point Exceptions**

Precision.

**Other Exceptions**

See Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

## CVTPD2DQ—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF E6 /r CVTPD2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert two packed double-precision floating-point values from <i>xmm2/m128</i> to two packed signed doubleword integers in <i>xmm1</i> .
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two signed doubleword integers in <i>xmm1</i> .
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ <i>xmm1, ymm2/m256</i>	RM	V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four signed doubleword integers in <i>xmm1</i> .

### Instruction Operand Encoding

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

### Description

Converts 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 XMM register. The result is stored in the low quadword of the destination operand and the high quadword is cleared to all 0s.

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.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:64) of the corresponding YMM register destination are zeroed.

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

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

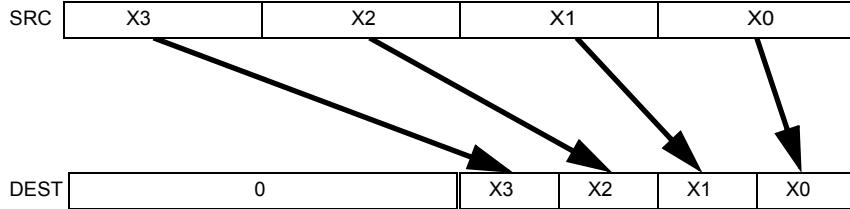


Figure 3-11. VCVTPD2DQ (VEX.256 encoded version)

## Operation

### 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[VLMAX-1:128] (unmodified)
```

### 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[VLMAX-1:64] ← 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[255:128] ← 0
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2DQ:    \_\_m128i \_mm\_cvtpd\_epi32 (\_\_m128d src)  
 CVTPD2DQ:    \_\_m128i \_mm256\_cvtpd\_epi32 (\_\_m256d src)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Other Exceptions

See Exceptions Type 2; additionally

#UD                      If VEX.vvvv != 1111B.

## CVTPD2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2D /r CVTPD2PI <i>mm, xmm/m128</i>		RM	Valid	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> .

### Instruction Operand Encoding

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

### Description

Converts 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 Table 22-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

**CVTPD2PS—Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Convert two packed double-precision floating-point values in <i>xmm2/m128</i> to two packed single-precision floating-point values in <i>xmm1</i> .
VEX.128.66.0F.WIG 5A /r VCVTPD2PS <i>xmm1, xmm2/m128</i>		RM V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two single-precision floating-point values in <i>xmm1</i> .
VEX.256.66.0F.WIG 5A /r VCVTPD2PS <i>xmm1, ymm2/m256</i>		RM V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four single-precision floating-point values in <i>xmm1</i> .

**Instruction Operand Encoding**

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

**Description**

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

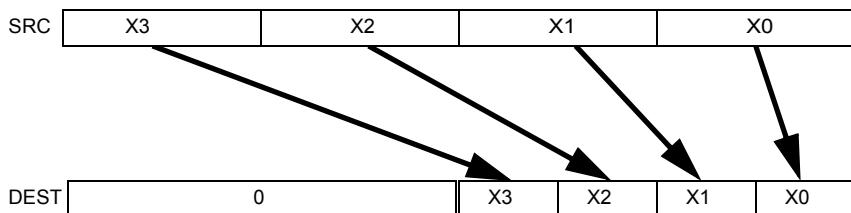
In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:64) of the corresponding YMM register destination are zeroed.

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

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



**Figure 3-12. VCVTPD2PS (VEX.256 encoded version)**

## Operation

### **CVTPD2PS (128-bit Legacy SSE version)**

DEST[31:0]  $\leftarrow$  Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[63:0])  
 DEST[63:32]  $\leftarrow$  Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[127:64])  
 DEST[127:64]  $\leftarrow$  0  
 DEST[VLMAX-1:128] (unmodified)

### **VCVTPD2PS (VEX.128 encoded version)**

DEST[31:0]  $\leftarrow$  Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[63:0])  
 DEST[63:32]  $\leftarrow$  Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[127:64])  
 DEST[VLMAX-1:64]  $\leftarrow$  0

### **VCVTPD2PS (VEX.256 encoded version)**

DEST[31:0]  $\leftarrow$  Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[63:0])  
 DEST[63:32]  $\leftarrow$  Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[127:64])  
 DEST[95:64]  $\leftarrow$  Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[191:128])  
 DEST[127:96]  $\leftarrow$  Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[255:192])  
 DEST[255:128]  $\leftarrow$  0

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2PS: `__m128 _mm_cvtpd_ps(__m128d a)`  
 CVTPD2PS: `__m256 _mm256_cvtpd_ps (__m256d a)`

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Other Exceptions

See Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

**CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values**

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2A /r CVTPI2PD <i>xmm, mm/m64*</i>	RM	Valid	Valid	Convert two packed signed doubleword integers from <i>mm/mem64</i> to two packed double-precision floating-point values in <i>xmm</i> .

**NOTES:**

\*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)	NA	NA

**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 Table 22-6, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

**CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values**

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2A /r CVTPI2PS <i>xmm, mm/m64</i>	RM	Valid	Valid	Convert two signed doubleword integers from <i>mm/m64</i> to two single-precision floating-point values in <i>xmm</i> .

**Instruction Operand Encoding**

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

**Description**

Converts 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 Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

**CVTPS2DQ—Convert Packed Single-Precision FP Values to Packed Dword Integers**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert four packed single-precision floating-point values from <i>xmm2/m128</i> to four packed signed doubleword integers in <i>xmm1</i> .
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed single precision floating-point values from <i>xmm2/mem</i> to four packed signed doubleword values in <i>xmm1</i> .
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Convert eight packed single precision floating-point values from <i>ymm2/mem</i> to eight packed signed doubleword values in <i>ymm1</i> .

**Instruction Operand Encoding**

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

**Description**

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

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

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

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

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

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

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

**Operation****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[VLMAX-1:128] (unmodified)
```

**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[VLMAX-1:128] ← 0
```

**VCVTPS2DQ (VEX.256 encoded version)**

DEST[31:0]  $\leftarrow$  Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32]  $\leftarrow$  Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64]  $\leftarrow$  Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96]  $\leftarrow$  Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[159:128]  $\leftarrow$  Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[159:128])  
 DEST[191:160]  $\leftarrow$  Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[191:160])  
 DEST[223:192]  $\leftarrow$  Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[223:192])  
 DEST[255:224]  $\leftarrow$  Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[255:224])

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

CVTPS2DQ: `__m128i _mm_cvtps_epi32(__m128 a)`  
 VCVTPS2DQ: `__m256i _mm256_cvtps_epi32 (__m256 a)`

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

See Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

**CVTPS2PD—Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values**

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

**Instruction Operand Encoding**

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

**Description**

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

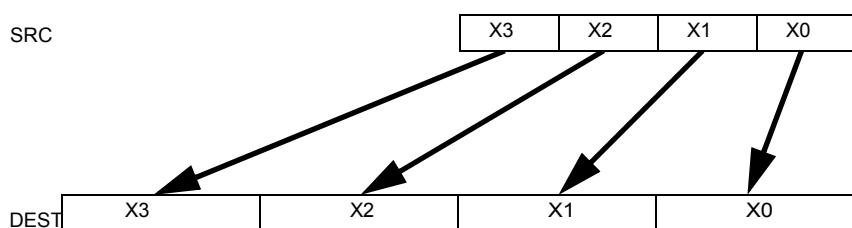
In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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



**Figure 3-13. CVTPS2PD (VEX.256 encoded version)**

## Operation

### CVTPS2PD (128-bit Legacy SSE version)

DEST[63:0]  $\leftarrow$  Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[127:64]  $\leftarrow$  Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[VLMAX-1:128] (unmodified)

### VCVTPS2PD (VEX.128 encoded version)

DEST[63:0]  $\leftarrow$  Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[127:64]  $\leftarrow$  Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### VCVTPS2PD (VEX.256 encoded version)

DEST[63:0]  $\leftarrow$  Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[127:64]  $\leftarrow$  Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[191:128]  $\leftarrow$  Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[255:192]  $\leftarrow$  Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[127:96])

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PD:    \_\_m128d \_mm\_cvtps\_pd(\_\_m128 a)  
 VCVTPS2PD:    \_\_m256d \_mm256\_cvtps\_pd (\_\_m128 a)

## SIMD Floating-Point Exceptions

Invalid, Denormal.

## Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

**CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers**

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2D /r CVTPS2PI <i>mm, xmm/m64</i>		RM	Valid	Convert two packed single-precision floating-point values from <i>xmm/m64</i> to two packed signed doubleword integers in <i>mm</i> .

**Instruction Operand Encoding**

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

**Description**

Converts 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 Table 22-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

## CVTSD2SI—Convert Scalar Double-Precision FP Value to Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 2D /r CVTSD2SI r32, xmm/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed doubleword integer <i>r32</i> .
F2 REX.W OF 2D /r CVTSD2SI r64, xmm/m64	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed quadword integer sign-extended into <i>r64</i> .
VEX.LIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed doubleword integer <i>r32</i> .
VEX.LIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64	RM	V/N.E. <sup>1</sup>	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed quadword integer sign-extended into <i>r64</i> .

### NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

### Instruction Operand Encoding

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

### Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

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

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. 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.

Legacy SSE instructions: Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

### Operation

IF 64-Bit Mode and OperandSize = 64

```

THEN
    DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer64(SRC[63:0]);
ELSE
    DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
FI;
```

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

```
int _mm_cvtsd_si32(__m128d a)  
__int64 _mm_cvtsd_si64(__m128d a)
```

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

See Exceptions Type 3; additionally

#UD                    If VEX.vvvv != 1111B.

**CVTSD2SS—Convert Scalar Double-Precision FP Value to Scalar Single-Precision FP Value**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 5A /r CVTSD2SS <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Convert one double-precision floating-point value in <i>xmm2/m64</i> to one single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.OF.WIG 5A /r VCVTSD2SS <i>xmm1,xmm2, xmm3/m64</i>	RVM	V/V	AVX	Convert one double-precision floating-point value in <i>xmm3/m64</i> to one single-precision floating-point value and merge with high bits in <i>xmm2</i> .

**Instruction Operand Encoding**

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

**Description**

Converts a double-precision floating-point value in the source operand (second operand) to a single-precision floating-point value 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 XMM register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand, and the upper 3 doublewords are left unchanged. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

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

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

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation****CVTSD2SS (128-bit Legacy SSE version)**

```
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[VLMAX-1:32] Unmodified *)
```

**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[VLMAX-1:128] ← 0
```

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

CVTSD2SS: `__m128_mm_cvtsd_ss(__m128 a, __m128d b)`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 3.

## CVTSI2SD—Convert Dword Integer to Scalar Double-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 2A /r CVTSI2SD <i>xmm, r/m32</i>	RM	V/V	SSE2	Convert one signed doubleword integer from <i>r/m32</i> to one double-precision floating-point value in <i>xmm</i> .
F2 REX.W OF 2A /r CVTSI2SD <i>xmm, r/m64</i>	RM	V/N.E.	SSE2	Convert one signed quadword integer from <i>r/m64</i> to one double-precision floating-point value in <i>xmm</i> .
VEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD <i>xmm1, xmm2, r/m32</i>	RVM	V/V	AVX	Convert one signed doubleword integer from <i>r/m32</i> to one double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD <i>xmm1, xmm2, r/m64</i>	RVM	V/N.E. <sup>1</sup>	AVX	Convert one signed quadword integer from <i>r/m64</i> to one double-precision floating-point value in <i>xmm1</i> .

### NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

### Instruction Operand Encoding

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

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Legacy SSE instructions: Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

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

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

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

**VCVTSI2SD**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

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

CVTSI2SD:     \_\_m128d \_mm\_cvtsi32\_sd(\_\_m128d a, int b)

CVTSI2SD:     \_\_m128d \_mm\_cvtsi64\_sd(\_\_m128d a, \_\_int64 b)

**SIMD Floating-Point Exceptions**

Precision.

**Other Exceptions**

See Exceptions Type 3.

## CVTSI2SS—Convert Dword Integer to Scalar Single-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 2A /r CVTSI2SS <i>xmm, r/m32</i>	RM	V/V	SSE	Convert one signed doubleword integer from <i>r/m32</i> to one single-precision floating-point value in <i>xmm</i> .
F3 REX.W OF 2A /r CVTSI2SS <i>xmm, r/m64</i>	RM	V/N.E.	SSE	Convert one signed quadword integer from <i>r/m64</i> to one single-precision floating-point value in <i>xmm</i> .
VEX.NDS.LIG.F3.OF.W0 2A /r VCVTSI2SS <i>xmm1, xmm2, r/m32</i>	RVM	V/V	AVX	Convert one signed doubleword integer from <i>r/m32</i> to one single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.W1 2A /r VCVTSI2SS <i>xmm1, xmm2, r/m64</i>	RVM	V/N.E. <sup>1</sup>	AVX	Convert one signed quadword integer from <i>r/m64</i> to one single-precision floating-point value in <i>xmm1</i> .

### NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

### Instruction Operand Encoding

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

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

**Legacy SSE instructions:** In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

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

**VEX.128 encoded version:** Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

#### 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]);
```

Fl;

DEST[VLMAX-1:32] (Unmodified)

**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[VLMAX-1:128] ← 0

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

CVTSI2SS:     \_\_m128 \_mm\_cvtsi32\_ss(\_\_m128 a, int b)

CVTSI2SS:     \_\_m128 \_mm\_cvtsi64\_ss(\_\_m128 a, \_\_int64 b)

**SIMD Floating-Point Exceptions**

Precision.

**Other Exceptions**

See Exceptions Type 3.

## CVTSS2SD—Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5A /r CVTSS2SD <i>xmm1, xmm2/m32</i>		RM V/V	SSE2	Convert one single-precision floating-point value in <i>xmm2/m32</i> to one double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.WIG 5A /r VCVTSS2SD <i>xmm1, xmm2, xmm3/m32</i>	RVM	V/V	AVX	Convert one single-precision floating-point value in <i>xmm3/m32</i> to one double-precision floating-point value and merge with high bits of <i>xmm2</i> .

### Instruction Operand Encoding

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

### Description

Converts a single-precision floating-point value in the source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand, and the high quadword is left unchanged.

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

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

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

#### CVTSS2SD (128-bit Legacy SSE version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[VLMAX-1:64] (Unmodified)
```

#### 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[VLMAX-1:128] ← 0
```

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

CVTSS2SD: `__m128d _mm_cvtss_sd(__m128d a, __m128 b)`

### SIMD Floating-Point Exceptions

Invalid, Denormal.

### Other Exceptions

See Exceptions Type 3.

**CVTSS2SI—Convert Scalar Single-Precision FP Value to Dword Integer**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32-bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
F3 OF 2D /r CVTSS2SI r32, xmm/m32	RM	V/V	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed doubleword integer in <i>r32</i> .
F3 REX.W OF 2D /r CVTSS2SI r64, xmm/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed quadword integer in <i>r64</i> .
VEX.LIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed doubleword integer in <i>r32</i> .
VEX.LIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32	RM	V/N.E. <sup>1</sup>	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed quadword integer in <i>r64</i> .

**NOTES:**

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

**Instruction Operand Encoding**

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

**Description**

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

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

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

**Legacy SSE instructions:** In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

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

**Operation**

IF 64-bit Mode and OperandSize = 64

THEN

  DEST[64:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

ELSE

  DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

FI;

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

```
int _mm_cvtss_si32(__m128d a)
__int64 _mm_cvtss_si64(__m128d a)
```

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

See Exceptions Type 3; additionally

#UD                  If VEX.vvvv != 1111B.

## CVTTPD2DQ—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Convert two packed double-precision floating-point values from <i>xmm2/m128</i> to two packed signed doubleword integers in <i>xmm1</i> using truncation.
VEX.128.66.0F.WIG E6 /r VCVTTPD2DQ <i>xmm1, xmm2/m128</i>		RM V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two signed doubleword integers in <i>xmm1</i> using truncation.
VEX.256.66.0F.WIG E6 /r VCVTTPD2DQ <i>xmm1, ymm2/m256</i>		RM V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four signed doubleword integers in <i>xmm1</i> using truncation.

### Instruction Operand Encoding

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

### Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

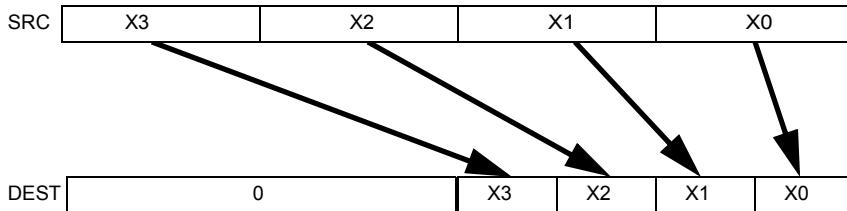
In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

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

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

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

**Figure 3-14. VCVTPD2DQ (VEX.256 encoded version)**

## Operation

### 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[VLMAX-1:128] (unmodified)
```

### VCVTPD2DQ (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[VLMAX-1:64] ← 0
```

### VCVTPD2DQ (VEX.256 encoded version)

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192])
DEST[255:128] ← 0
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD2DQ: `__m128i _mm_cvttpd_epi32(__m128d a)`  
 VCVTPD2DQ: `__m128i _mm256_cvttpd_epi32 (__m256d src)`

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

## CVTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /r CVTPD2PI <i>mm</i> , <i>xmm/m128</i>		RM	Valid	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> using truncation.

### Instruction Operand Encoding

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

### Description

Converts 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 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_Truncate(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32_
Truncate(SRC[127:64]);
```

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

CVTPD1PI:    \_\_m64 \_mm\_cvtpd\_pi32(\_\_m128d a)

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Mode Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

## CVTTPS2DQ—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5B /r CVTTPS2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert four single-precision floating-point values from <i>xmm2/m128</i> to four signed doubleword integers in <i>xmm1</i> using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed single precision floating-point values from <i>xmm2/mem</i> to four packed signed doubleword values in <i>xmm1</i> using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Convert eight packed single precision floating-point values from <i>ymm2/mem</i> to eight packed signed doubleword values in <i>ymm1</i> using truncation.

### Instruction Operand Encoding

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

### Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

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

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

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

### Operation

#### 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[VLMAX-1:128] (unmodified)
```

**VCVTPPS2DQ (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[VLMAX-1:128] ← 0
```

**VCVTPPS2DQ (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])
```

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

`CVTPPS2DQ: __m128i _mm_cvttps_epi32(__m128 a)`  
`VCVTPPS2DQ: __m256i _mm256_cvttps_epi32 (__m256 a)`

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

See Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

## CVTTPS2PI—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2C /r CVTTPS2PI <i>mm, xmm/m64</i>	RM	Valid	Valid	Convert two single-precision floating-point values from <i>xmm/m64</i> to two signed doubleword signed integers in <i>mm</i> using truncation.

### Instruction Operand Encoding

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

### Description

Converts 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 Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

**CVTTSD2SI—Convert with Truncation Scalar Double-Precision FP Value to Signed Integer**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 2C /r CVTTSD2SI r32, xmm/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed doubleword integer in <i>r32</i> using truncation.
F2 REX.W OF 2C /r CVTTSD2SI r64, xmm/m64	RM	V/N.E.	SSE2	Convert one double precision floating-point value from <i>xmm/m64</i> to one signedquadword integer in <i>r64</i> using truncation.
VEX.LIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from <i>xmm1/m64</i> to one signed doubleword integer in <i>r32</i> using truncation.
VEX.LIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64	RM	V/N.E. <sup>1</sup>	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed quadword integer in <i>r64</i> using truncation.

**NOTES:**

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

**Instruction Operand Encoding**

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

**Description**

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

**Operation**

IF 64-Bit Mode and OperandSize = 64

THEN

```
DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_
    Integer64_Truncate(SRC[63:0]);
```

ELSE

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_
    Integer32_Truncate(SRC[63:0]);
```

FI;

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

```
int _mm_cvtsd_si32(_m128d a)
__int64 _mm_cvtsd_si64(_m128d a)
```

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Other Exceptions

See Exceptions Type 3; additionally

#UD                    If VEX.vvvv != 1111B.

**CVTTSS2SI—Convert with Truncation Scalar Single-Precision FP Value to Dword Integer**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 2C /r CVTTSS2SI r32, xmm/m32	RM	V/V	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed doubleword integer in <i>r32</i> using truncation.
F3 REX.W OF 2C /r CVTTSS2SI r64, xmm/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed quadword integer in <i>r64</i> using truncation.
VEX.LIG.F3.OF.W0 2C /r VCVTTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed doubleword integer in <i>r32</i> using truncation.
VEX.LIG.F3.OF.W1 2C /r VCVTTSS2SI r64, xmm1/m32	RM	V/N.E. <sup>1</sup>	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed quadword integer in <i>r64</i> using truncation.

**NOTES:**

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

**Instruction Operand Encoding**

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

**Description**

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

**Operation**

IF 64-Bit Mode and OperandSize = 64

THEN

```
DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_
Integer_Truncate(SRC[31:0]);
```

ELSE

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_
Integer_Truncate(SRC[31:0]);
```

FI;

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

```
int _mm_cvttss_si32(_m128d a)
__int64 _mm_cvttss_si64(_m128d a)
```

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

See Exceptions Type 3; additionally

#UD                  If VEX.vvvv != 1111B.

**CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
99	CWD		NP	Valid	DX:AX ← sign-extend of AX.
99	CDQ		NP	Valid	EDX:EAX ← sign-extend of EAX.
REX.W + 99	CQO		NP	Valid	N.E. RDX:RAX← sign-extend of RAX.

**Instruction Operand Encoding**

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

**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); Fl;
  ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
    RDX ← SignExtend(RAX); Fl;
Fl;
```

**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	NP	Invalid	Valid	Decimal adjust AL after addition.

### Instruction Operand Encoding

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

### 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	NP	Invalid	Valid	Decimal adjust AL after subtraction.

### Instruction Operand Encoding

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

### 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	M	Valid	Valid	Decrement r/m8 by 1.
REX + FE /1	DEC r/m8*	M	Valid	N.E.	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:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

### 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	M	Valid	Valid	Unsigned divide AX by r/m8, with result stored in AL ← Quotient, AH ← Remainder.
REX + F6 /6	DIV r/m8*	M	Valid	N.E.	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:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

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

### 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 3-25.

Table 3-25. 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; Fl; (* 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;
        Fl;
    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;
            Fl;
        Fl;
    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;
            Fl;
        Fl;
    ELSE IF 64-Bit Mode and Operandsize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp ← RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFF
                THEN #DE; (* Divide error *)
            ELSE
                RAX ← temp;
                RDX ← RDX:RAX MOD SRC;
            Fl;
        Fl;
    Fl;

```

## 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	CPUID Feature Flag	Description
66 0F 5E /r DIVPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Divide packed double-precision floating-point values in <i>xmm1</i> by packed double-precision floating-point values <i>xmm2/m128</i> .
VEX.NDS.128.66.0F.WIG 5E /r VDIVPD <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Divide packed double-precision floating-point values in <i>xmm2</i> by packed double-precision floating-point values in <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5E /r VDIVPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Divide packed double-precision floating-point values in <i>ymm2</i> by packed double-precision floating-point values in <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

**Description**

Performs an SIMD divide of the two or four packed double-precision floating-point values in the first source operand by the two or four packed double-precision floating-point values in the second source operand. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a SIMD double-precision floating-point operation.

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

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

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

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

**Operation****DIVPD (128-bit Legacy SSE version)**

DEST[63:0]  $\leftarrow$  SRC1[63:0] / SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64] / SRC2[127:64]  
 DEST[VLMAX-1:128] (Unmodified)

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

DEST[63:0]  $\leftarrow$  SRC1[63:0] / SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64] / SRC2[127:64]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

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

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

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

DIVPD:    `__m128d _mm_div_pd(__m128d a, __m128d b)`  
 VDIVPD:    `__m256d _mm256_div_pd (__m256d a, __m256d b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2.

**DIVPS—Divide Packed Single-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5E /r DIVPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Divide packed single-precision floating-point values in <i>xmm1</i> by packed single-precision floating-point values <i>xmm2/m128</i> .
VEX.NDS.128.0F.WIG 5E /r VDIVPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Divide packed single-precision floating-point values in <i>xmm2</i> by packed double-precision floating-point values in <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 5E /r VDIVPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Divide packed single-precision floating-point values in <i>ymm2</i> by packed double-precision floating-point values in <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

**Description**

Performs an SIMD divide of the four or eight packed single-precision floating-point values in the first source operand by the four or eight packed single-precision floating-point values in the second source operand. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a SIMD single-precision floating-point operation.

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

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

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

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

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

**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[VLMAX-1:128] ← 0
```

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

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] / \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] / \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] / \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] / \text{SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] / \text{SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] / \text{SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] / \text{SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] / \text{SRC2}[255:224].$

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

DIVPS:     `__m128 _mm_div_ps(__m128 a, __m128 b)`  
 VDIVPS:    `__m256 _mm256_div_ps (__m256 a, __m256 b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2.

**DIVSD—Divide Scalar Double-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 5E /r DIVSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Divide low double-precision floating-point value in <i>xmm1</i> by low double-precision floating-point value in <i>xmm2/mem64</i> .
VEX.NDS.LIG.F2.0F.WIG 5E /r VDIVSD <i>xmm1, xmm2, xmm3/m64</i>	RVM	V/V	AVX	Divide low double-precision floating point values in <i>xmm2</i> by low double precision floating-point value in <i>xmm3/mem64</i> .

**Instruction Operand Encoding**

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

**Description**

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination hyperons are XMM registers. The high quadword of the destination operand is copied from the high quadword of the first source operand. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a scalar double-precision floating-point operation.

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

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

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

**Operation****DIVSD (128-bit Legacy SSE version)**

$DEST[63:0] \leftarrow DEST[63:0] / SRC[63:0]$   
 $DEST[VLMAX-1:64] \text{ (Unmodified)}$

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

$DEST[63:0] \leftarrow SRC1[63:0] / SRC2[63:0]$   
 $DEST[127:64] \leftarrow SRC1[127:64]$   
 $DEST[VLMAX-1:128] \leftarrow 0$

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

DIVSD: `__m128d _mm_div_sd (m128d a, m128d b)`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 3.

## DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5E /r DIVSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Divide low single-precision floating-point value in <i>xmm1</i> by low single-precision floating-point value in <i>xmm2/m32</i> .
VEX.NDS.LIG.F3.0F.WIG 5E /r VDIVSS <i>xmm1, xmm2, xmm3/m32</i>	RVM	V/V	AVX	Divide low single-precision floating point value in <i>xmm2</i> by low single precision floating-point value in <i>xmm3/m32</i> .

### Instruction Operand Encoding

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

### Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. The three high-order doublewords of the destination are copied from the same dwords of the first source operand. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a scalar single-precision floating-point operation.

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

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

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

### Operation

#### DIVSS (128-bit Legacy SSE version)

DEST[31:0]  $\leftarrow$  DEST[31:0] / SRC[31:0]  
DEST[VLMAX-1:32] (Unmodified)

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

DEST[31:0]  $\leftarrow$  SRC1[31:0] / SRC2[31:0]  
DEST[127:32]  $\leftarrow$  SRC1[127:32]  
DEST[VLMAX-1:128]  $\leftarrow$  0

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

DIVSS: \_\_m128 \_mm\_div\_ss(\_\_m128 a, \_\_m128 b)

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

#### Other Exceptions

See Exceptions Type 3.

**DPPD – Dot Product of Packed Double Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 41 /r ib DPPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Selectively multiply packed DP floating-point values from <i>xmm1</i> with packed DP floating-point values from <i>xmm2</i> , add and selectively store the packed DP floating-point values to <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 41 /r ib VDPPD <i>xmm1,xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Selectively multiply packed DP floating-point values from <i>xmm2</i> with packed DP floating-point values from <i>xmm3</i> , add and selectively store the packed DP floating-point values to <i>xmm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

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

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

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; Fl;
IF (imm8[5] = 1)
    THEN Temp1[127:64] ← DEST[127:64] * SRC[127:64]; // update SIMD exception flags
    ELSE Temp1[127:64] ← +0.0; Fl;
/* 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; Fl;
IF (imm8[1] = 1)
    THEN DEST[127:64] ← Temp2[63:0];
    ELSE DEST[127:64] ← +0.0; Fl;

```

### DPPD (128-bit Legacy SSE version)

```

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

```

### VDPPD (VEX.128 encoded version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-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 Exceptions Type 2; 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 <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Selectively multiply packed SP floating-point values from <i>xmm1</i> with packed SP floating-point values from <i>xmm2</i> , add and selectively store the packed SP floating-point values or zero values to <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 40 /r ib VDPPS <i>xmm1,xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Multiply packed SP floating point values from <i>xmm1</i> with packed SP floating point values from <i>xmm2/mem</i> selectively add and store to <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 40 /r ib VDPPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Multiply packed single-precision floating-point values from <i>ymm2</i> with packed SP floating point values from <i>ymm3/mem</i> , selectively add pairs of elements and store to <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

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

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

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

## Operation

```

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; Fl;
IF (imm8[5] = 1)
    THEN Temp1[63:32] ← DEST[63:32] * SRC[63:32]; // update SIMD exception flags
    ELSE Temp1[63:32] ← +0.0; Fl;
IF (imm8[6] = 1)
    THEN Temp1[95:64] ← DEST[95:64] * SRC[95:64]; // update SIMD exception flags
    ELSE Temp1[95:64] ← +0.0; Fl;
IF (imm8[7] = 1)
    THEN Temp1[127:96] ← DEST[127:96] * SRC[127:96]; // update SIMD exception flags
    ELSE Temp1[127:96] ← +0.0; Fl;

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; Fl;
IF (imm8[1] = 1)
    THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0; Fl;
IF (imm8[2] = 1)
    THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0; Fl;
IF (imm8[3] = 1)
    THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0; Fl;

```

### DPPS (128-bit Legacy SSE version)

DEST[127:0] ← DP\_Primitive(SRC1[127:0], SRC2[127:0]);  
 DEST[VLMAX-1:128] (Unmodified)

### VDPPS (VEX.128 encoded version)

DEST[127:0] ← DP\_Primitive(SRC1[127:0], SRC2[127:0]);  
 DEST[VLMAX-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 Exceptions Type 2.

## EMMS—Empty MMX Technology State

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 77	EMMS	NP	Valid	Valid	Set the x87 FPU tag word to empty.

### Instruction Operand Encoding

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

### 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 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 CRO.EM[bit 2] = 1.
#NM	If CRO.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.

## 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 nested stack frame for a procedure.
C8 iw ib	ENTER imm16, imm8	II	Valid	Valid	Create a nested stack frame for a procedure.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
II	iw	imm8	NA	NA

### Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits), EBP (32 bits), or RBP (64 bits) register specifies the current frame pointer and whether SP (16 bits), ESP (32 bits), or RSP (64 bits) specifies the stack pointer. In 64-bit mode, stack-size attribute is always 64-bits.

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.

### Operation

NestingLevel  $\leftarrow$  NestingLevel MOD 32

IF 64-Bit Mode (StackSize = 64)

THEN

Push(RBP);

FrameTemp  $\leftarrow$  RSP;

ELSE IF StackSize = 32

THEN

Push(EBP);

FrameTemp  $\leftarrow$  ESP; FI;

ELSE (\* StackSize = 16 \*)

Push(BP);

FrameTemp  $\leftarrow$  SP;

FI;

IF NestingLevel = 0

## INSTRUCTION SET REFERENCE, A-M

```
THEN GOTO CONTINUE;
Fl;

IF (NestingLevel > 1)
    THEN FOR i ← 1 to (NestingLevel - 1)
        DO
            IF 64-Bit Mode (StackSize = 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 *)
                        Fl;
                    Fl;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 32
                        THEN
                            EBP ← EBP - 2;
                            Push([EBP]); (* Word push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 2;
                            Push([BP]); (* Word push *)
                        Fl;
                    Fl;
                OD;
            Fl;

IF 64-Bit Mode (StackSize = 64)
    THEN
        Push(FrameTemp); (* Quadword push *)
    ELSE IF OperandSize = 32
        THEN
            Push(FrameTemp); Fl; (* Doubleword push *)
    ELSE (* OperandSize = 16 *)
        Push(FrameTemp); (* Word push *)
    Fl;

CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
        RBP ← FrameTemp;
        RSP ← RSP - Size;
    ELSE IF StackSize = 32
        THEN
            EBP ← FrameTemp;
            ESP ← ESP - Size; Fl;
    ELSE (* StackSize = 16 *)
        BP ← FrameTemp;
        SP ← SP - Size;
```

Fl;  
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.

## EXTRACTPS – Extract Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm2, imm8	MRI	V/V	SSE4_1	Extract a single-precision floating-point value from <i>xmm2</i> at the source offset specified by <i>imm8</i> and store the result to <i>reg</i> or <i>m32</i> . The upper 32 bits of r64 is zeroed if reg is r64.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS r/m32, xmm1, imm8	MRI	V/V	AVX	Extract one single-precision floating-point value from <i>xmm1</i> at the offset specified by <i>imm8</i> and store the result in <i>reg</i> or <i>m32</i> . Zero extend the results in 64-bit register if applicable.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

### Description

Extracts a single-precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single-precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

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.

VEX.128 encoded version: When VEX.128.66.0F3A.W1 17 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits. VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

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

#### 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 OFFFFFFFh
    DEST[63:32] ← 0
ELSE
    DEST[31:0] ← (SRC[127:0] » (SRC_OFFSET*32)) AND OFFFFFFFh
FI

```

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

```

SRC_OFFSET ← IMM8[1:0]
IF ( 64-Bit Mode and DEST is register)
    DEST[31:0] ← (SRC[127:0] » (SRC_OFFSET*32)) AND 0xFFFFFFFFh
    DEST[63:32] ← 0
ELSE
    DEST[31:0] ← (SRC[127:0] » (SRC_OFFSET*32)) AND 0xFFFFFFFFh
FI

```

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

EXTRACTPS:    `_mm_extractmem_ps (float *dest, __m128 a, const int nidx);`

EXTRACTPS:    `__m128 _mm_extract_ps (__m128 a, const int nidx);`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 5; additionally

#UD                If VEX.L= 1.

## F2XM1—Compute $2^{ST(0)} - 1$

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 3-26. 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:

$$x^y \leftarrow 2^{(y * \log_2 x)}$$

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

### Operation

$$ST(0) \leftarrow (2^{ST(0)} - 1);$$

### FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.
- CO, 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 CRO.EM[bit 2] or CRO.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 3-27. 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**

$ST(0) \leftarrow |ST(0)|;$

**FPU Flags Affected**

C1 Set to 0.  
C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow occurred.

**Protected Mode Exceptions**

#NM CRO.EM[bit 2] or CRO.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 <i>m32fp</i>	Valid	Valid	Add <i>m32fp</i> to ST(0) and store result in ST(0).
DC /0	FADD <i>m64fp</i>	Valid	Valid	Add <i>m64fp</i> 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 <i>m32int</i>	Valid	Valid	Add <i>m32int</i> to ST(0) and store result in ST(0).
DE /0	FIADD <i>m16int</i>	Valid	Valid	Add <i>m16int</i> 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  $\infty$  of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-28.

Table 3-28. 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 + Convert.ToDoubleExtendedPrecisionFP(SRC);

ELSE (\* Source operand is floating-point value \*)

DEST ← DEST + SRC;

Fl;

IF Instruction = FADDP

THEN

PopRegisterStack;

Fl;

**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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m80dec</i>	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) ← Convert.ToDoubleExtendedPrecisionFP(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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 3-29. 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, $\pm\infty$ , 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 3-30. FCHS Results

ST(0) SRC	ST(0) DEST
- $\infty$	+ $\infty$
- F	+ F
- 0	+ 0
+ 0	- 0
+ F	- F
+ $\infty$	- $\infty$
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

$\text{SignBit}(\text{ST}(0)) \leftarrow \text{NOT}(\text{SignBit}(\text{ST}(0)))$ ;

### FPU Flags Affected

C1                    Set to 0.  
 C0, C2, C3        Undefined.

### Floating-Point Exceptions

#IS                  Stack underflow occurred.

### Protected Mode Exceptions

#NM                  CRO.EM[bit 2] or CRO.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*	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*	Valid	Valid	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

### NOTES:

\* 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 a Pentium 4, Intel Xeon, or P6 family processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCRS 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	CRO.EM[bit 2] or CRO.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.

## FCMOVcc—Floating-Point Conditional Move

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode*	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	FCMOVNBT 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:

\* 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 is 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 FCMOVcc 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 FCMOVcc instructions. Software can check if the FCMOVcc 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 FCMOVcc 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                    CRO.EM[bit 2] or CRO.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 <i>m32fp</i>	Valid	Valid	Compare ST(0) with <i>m32fp</i> .
DC /2	FCOM <i>m64fp</i>	Valid	Valid	Compare ST(0) with <i>m64fp</i> .
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 <i>m32fp</i>	Valid	Valid	Compare ST(0) with <i>m32fp</i> and pop register stack.
DC /3	FCOMP <i>m64fp</i>	Valid	Valid	Compare ST(0) with <i>m64fp</i> 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 3-31. 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 ModR/M” 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 = FCMP
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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 3-32. 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 ModR/M” 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               CRO.EM[bit 2] or CRO.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 cosine.

### Description

Computes the 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 3-33. 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  $2\pi$  or by using the FPREM instruction with a divisor of  $2\pi$ . See the section titled “Pi” 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  $\pi$  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  $\leftarrow$  0;
  ST(0)  $\leftarrow$  cosine(ST(0));
ELSE (* Source operand is out-of-range *)
  C2  $\leftarrow$  1;
FI;
```

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C2	Undefined if C2 is 1. Set to 1 if outside range ( $-2^{63} < \text{source operand} < +2^{63}$ ); otherwise, set to 0.
C0, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Source is a denormal value.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	CRO.EM[bit 2] or CRO.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                   CRO.EM[bit 2] or CRO.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 <i>m32fp</i>	Valid	Valid	Divide ST(0) by <i>m32fp</i> and store result in ST(0).
DC /6	FDIV <i>m64fp</i>	Valid	Valid	Divide ST(0) by <i>m64fp</i> 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 <i>m32int</i>	Valid	Valid	Divide ST(0) by <i>m32int</i> and store result in ST(0).
DE /6	FIDIV <i>m16int</i>	Valid	Valid	Divide ST(0) by <i>m16int</i> 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  $\infty$  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 3-34. FDIV/FDIVP/FIDIV Results

		DEST						
		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
SRC	$-\infty$	*	$+0$	$+0$	$-0$	$-0$	*	NaN
	$-F$	$+\infty$	$+F$	$+0$	$-0$	$-F$	$-\infty$	NaN
	$-I$	$+\infty$	$+F$	$+0$	$-0$	$-F$	$-\infty$	NaN
	$-0$	$+\infty$	**	*	*	**	$-\infty$	NaN
	$+0$	$-\infty$	**	*	*	**	$+\infty$	NaN
	$+I$	$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$+F$	$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$+\infty$	*	$-0$	$-0$	$+0$	$+0$	*	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 / Convert.ToDoubleExtendedPrecisionFP(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.  
 $\pm\infty / \pm\infty$ ;  $\pm0 / \pm0$   
#D Source is a denormal value.

#Z	DEST / $\pm 0$ , where DEST is not equal to $\pm 0$ .
#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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m32fp</i>	Valid	Valid	Divide <i>m32fp</i> by ST(0) and store result in ST(0).
DC /7	FDIVR <i>m64fp</i>	Valid	Valid	Divide <i>m64fp</i> by ST(0) and store result in ST(0).
D8 F8+i	FDIVR ST(i), ST(0)	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 <i>m32int</i>	Valid	Valid	Divide <i>m32int</i> by ST(0) and store result in ST(0).
DE /7	FIDIVR <i>m16int</i>	Valid	Valid	Divide <i>m16int</i> 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  $\infty$  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 3-35. 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 ← Convert.ToDoubleExtendedPrecisionFP(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 / $\pm 0$ , where SRC is not equal to $\pm 0$ .
#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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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

$\text{TAG}(i) \leftarrow 11\text{B};$

### FPU Flags Affected

C0, C1, C2, C3 undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

- #NM                    CRO.EM[bit 2] or CRO.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 <i>m16int</i>	Valid	Valid	Compare ST(0) with <i>m16int</i> .
DA /2	FICOM <i>m32int</i>	Valid	Valid	Compare ST(0) with <i>m32int</i> .
DE /3	FICOMP <i>m16int</i>	Valid	Valid	Compare ST(0) with <i>m16int</i> and pop stack register.
DA /3	FICOMP <i>m32int</i>	Valid	Valid	Compare ST(0) with <i>m32int</i> 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 3-36. 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 ModR/M” 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  $-0.0 \leftarrow +0.0$ .

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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m16int</i>	Valid	Valid	Push <i>m16int</i> onto the FPU register stack.
DB /0	FILD <i>m32int</i>	Valid	Valid	Push <i>m32int</i> onto the FPU register stack.
DF /5	FILD <i>m64int</i>	Valid	Valid	Push <i>m64int</i> 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) ← Convert.ToDoubleExtendedPrecisionFP(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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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                    CRO.EM[bit 2] or CRO.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*	Valid	Valid	Initialize FPU without checking for pending unmasked floating-point exceptions.

### NOTES:

\* 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 a Pentium 4, Intel Xeon, or P6 family 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	CRO.EM[bit 2] or CRO.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 <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> .
DB /2	FIST <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> .
DF /3	FISTP <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> and pop register stack.
DB /3	FISTP <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> and pop register stack.
DF /7	FISTP <i>m64int</i>	Valid	Valid	Store ST(0) in <i>m64int</i> 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 3-37. FIST/FISTP Results**

ST(0)	DEST
-∞ 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
+∞ 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  $\pm 1$ , 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  $\infty$ , 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.
C0, C2, C3	Set to 0 otherwise. Undefined.

## Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value is too large for the destination format.
#P	Source operand is an SNaN, QNaN, $\pm\infty$ , or unsupported format. 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 3-38. FISTTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	-I
$-1 < F < +1$	0
$F \geq +1$	+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.

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

### Operation

DEST  $\leftarrow$  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 CRO.EM[bit 2] = 1. If CRO.TS[bit 3] = 1.

#UD                    If CPUID.01H: ECX.SSE3[bit 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.  
                       If CRO.EM[bit 2] = 1.  
                       If CRO.TS[bit 3] = 1.  
                       If CPUID.01H: ECX.SSE3[bit 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.  
                       If CRO.EM[bit 2] = 1.  
                       If CRO.TS[bit 3] = 1.  
                       If CPUID.01H: ECX.SSE3[bit 0] = 0.  
                       If the LOCK prefix is used.  
                       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            CRO.EM[bit 2] or CRO.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 <i>m32fp</i>	Valid	Valid	Push <i>m32fp</i> onto the FPU register stack.
DD /0	FLD <i>m64fp</i>	Valid	Valid	Push <i>m64fp</i> onto the FPU register stack.
DB /5	FLD <i>m80fp</i>	Valid	Valid	Push <i>m80fp</i> 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.

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);
  Fl;
TOP ← TOP – 1;

IF SRC is memory-operand
  THEN
    ST(0) ← Convert.ToDoubleExtendedPrecisionFP(SRC);
  ELSE (* SRC is ST(i) *)
    ST(0) ← temp;
  Fl;

```

### 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 $\pi$ 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,  $\log_2 10$ ,  $\log_2 e$ ,  $\pi$ ,  $\log_{10} 2$ , 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 “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the  $\pi$  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	CRO.EM[bit 2] or CRO.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 <i>m2byte</i> .

### 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m14/28byte</i>	Valid	Valid	Load FPU environment from <i>m14byte</i> or <i>m28byte</i> .

### 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	CRO.EM[bit 2] or CRO.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 CRO.EM[bit 2] or CRO.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 CRO.EM[bit 2] or CRO.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 CRO.EM[bit 2] or CRO.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 <i>m32fp</i>	Valid	Valid	Multiply ST(0) by <i>m32fp</i> and store result in ST(0).
DC /1	FMUL <i>m64fp</i>	Valid	Valid	Multiply ST(0) by <i>m64fp</i> 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 <i>m32int</i>	Valid	Valid	Multiply ST(0) by <i>m32int</i> and store result in ST(0).
DE /1	FIMUL <i>m16int</i>	Valid	Valid	Multiply ST(0) by <i>m16int</i> 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  $\infty$ . 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 3-39. 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 \* Convert.ToDoubleExtendedPrecisionFP(SRC);

ELSE (\* Source operand is floating-point value \*)

DEST ← DEST \* SRC;

Fl;

IF Instruction = FMULP

THEN

PopRegisterStack;

Fl;

**FPU Flags Affected**

C1                         Set to 0 if stack underflow occurred.

Set if result was rounded up; cleared otherwise.

CO, 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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.

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 | CRO.EM[bit 2] or CRO.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*	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:

\* 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  $+\pi$ .

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  $\pi/2$  and  $\pi$ , while a point  $(X, -Y)$  is in the fourth quadrant, resulting in an angle between 0 and  $-\pi/2$ . A point  $(-X, -Y)$  is in the third quadrant, giving an angle between  $-\pi/2$  and  $-\pi$ .

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

**Table 3-40. FPATAN Results**

		ST(0)						
		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
ST(1)	$-\infty$	$-3\pi/4^*$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4^*$	NaN
	$-F$	$-p$	$-\pi$ to $-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$ to $-0$	$-0$	NaN
	$-0$	$-p$	$-p$	$-p^*$	$-0^*$	$-0$	$-0$	NaN
	$+0$	$+p$	$+p$	$+\pi^*$	$+0^*$	$+0$	$+0$	NaN
	$+F$	$+p$	$+\pi$ to $+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$ to $+0$	$+0$	NaN
	$+\infty$	$+3\pi/4^*$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/4^*$	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  $\infty/\infty$  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  $\infty/\infty$  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,  $\text{arctangent}(0,0)$  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:

$$0 \leq |ST(1)| < |ST(0)| < +\infty$$

**Operation**

$ST(1) \leftarrow \arctan(ST(1) / ST(0));$   
 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	CRO.EM[bit 2] or CRO.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:

$$\text{Remainder} \leftarrow \text{ST}(0) - (\text{Q} * \text{ST}(1))$$

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 3-41. 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.

\*\* Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is  $\infty$ , 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  $\pi/4$ ), 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 $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

## Protected Mode Exceptions

#NM	CRO.EM[bit 2] or CRO.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:

$$\text{Remainder} \leftarrow \text{ST}(0) - (\text{Q} * \text{ST}(1))$$

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 3-42. 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.

\*\* Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is  $\infty$ , 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  $\pi/4$ ), 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 $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

## Protected Mode Exceptions

#NM	CRO.EM[bit 2] or CRO.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 tangent and push 1 onto the FPU stack.

### Description

Computes the 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 3-43. 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  $2\pi$  or by using the FPREM instruction with a divisor of  $2\pi$ . See the section titled “Pi” 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  $\pi$  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) ← tan(ST(0));
    TOP ← TOP - 1;
    ST(0) ← 1.0;
  ELSE (* Source operand is out-of-range *)
    C2 ← 1;
  Fl;
```

**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 ( $-2^{63} < \text{source operand} < +2^{63}$ ); otherwise, set to 0.
C0, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , 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	CRO.EM[bit 2] or CRO.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  $\infty$ , 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

$ST(0) \leftarrow \text{RoundToIntegralValue}(ST(0));$

### 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	CRO.EM[bit 2] or CRO.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 <i>m94/108byte</i>	Valid	Valid	Load FPU state from <i>m94byte</i> or <i>m108byte</i> .

### 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. 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, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

### Protected Mode Exceptions

- |        |  |
|--------|--|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m94/108byte</i>	Valid	Valid	Store FPU state to <i>m94byte</i> or <i>m108byte</i> after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE* <i>m94/108byte</i>	Valid	Valid	Store FPU environment to <i>m94byte</i> or <i>m108byte</i> without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

### NOTES:

\* 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 a Pentium 4, Intel Xeon, or P6 family processor.

## Operation

(\* Save FPU State and Registers \*)

```
DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUIstructionPointer] ← FPUIstructionPointer;
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;
FPUIstructionPointer ← 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 3-44. 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							

#### 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

$$ST(0) \leftarrow ST(0) * 2^{\text{RoundTowardZero}(ST(1))};$$

**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	CRO.EM[bit 2] or CRO.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 its sine.

### Description

Computes 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 3-45. FSIN Results

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
$-F$	$-1 \text{ to } +1$
$-0$	$-0$
$+0$	$+0$
$+F$	$-1 \text{ to } +1$
$+\infty$	*
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  $2\pi$  or by using the FPREM instruction with a divisor of  $2\pi$ . See the section titled “Pi” 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  $\pi$  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  $\leftarrow 0$ ;
    ST(0)  $\leftarrow \sin(ST(0))$ ;
  ELSE (* Source operand out of range *)
    C2  $\leftarrow 1$ ;
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 ( $-2^{63} < \text{source operand} < +2^{63}$ ); otherwise, set to 0.
C0, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	CRO.EM[bit 2] or CRO.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 sine, and push the cosine onto the register stack.

### Description

Computes both the 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 3-46. FSINCOS Results

SRC	DEST	
	ST(1) Cosine	ST(0) Sine
$-\infty$	*	*
$-F$	$-1 \text{ to } +1$	$-1 \text{ to } +1$
$-0$	$+1$	$-0$
$+0$	$+1$	$+0$
$+F$	$-1 \text{ to } +1$	$-1 \text{ to } +1$
$+\infty$	*	*
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  $2\pi$  or by using the FPREM instruction with a divisor of  $2\pi$ . See the section titled “PI” 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  $\pi$  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  $\leftarrow$  0;
    TEMP  $\leftarrow$  cosine(ST(0));
    ST(0)  $\leftarrow$  sine(ST(0));
    TOP  $\leftarrow$  TOP - 1;
    ST(0)  $\leftarrow$  TEMP;
  ELSE (* Source operand out of range *)
    C2  $\leftarrow$  1;
  Fi;

```

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurs.
	Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ( $-2^{63} < \text{source operand} < +2^{63}$ ); otherwise, set to 0.
C0, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , 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	CRO.EM[bit 2] or CRO.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 3-47. 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

$ST(0) \leftarrow \text{SquareRoot}(ST(0));$

### 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	$CRO.EM[bit 2]$ or $CRO.TS[bit 3] = 1$ .
#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 <i>m32fp</i>	Valid	Valid	Copy ST(0) to <i>m32fp</i> .
DD /2	FST <i>m64fp</i>	Valid	Valid	Copy ST(0) to <i>m64fp</i> .
DD D0+i	FST ST(i)	Valid	Valid	Copy ST(0) to ST(i).
D9 /3	FSTP <i>m32fp</i>	Valid	Valid	Copy ST(0) to <i>m32fp</i> and pop register stack.
DD /3	FSTP <i>m64fp</i>	Valid	Valid	Copy ST(0) to <i>m64fp</i> and pop register stack.
DB /7	FSTP <i>m80fp</i>	Valid	Valid	Copy ST(0) to <i>m80fp</i> 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  $\pm 0$ ,  $\pm \infty$ , 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,  $\infty$ , 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.<br>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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m2byte</i>	Valid	Valid	Store FPU control word to <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW* <i>m2byte</i>	Valid	Valid	Store FPU control word to <i>m2byte</i> without checking for pending unmasked floating-point exceptions.

### NOTES:

\* 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 a Pentium 4, Intel Xeon, or P6 family 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m14/28byte</i>	Valid	Valid	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV* <i>m14/28byte</i>	Valid	Valid	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

### NOTES:

\* 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 a Pentium 4, Intel Xeon, or P6 family processor.

### Operation

```
DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUIInstructionPointer] ← FPUIInstructionPointer;
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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m2byte</i>	Valid	Valid	Store FPU status word at <i>m2byte</i> 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* <i>m2byte</i>	Valid	Valid	Store FPU status word at <i>m2byte</i> without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW* AX	Valid	Valid	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

### NOTES:

\* 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 a Pentium 4, Intel Xeon, or P6 family 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m32fp</i>	Valid	Valid	Subtract <i>m32fp</i> from ST(0) and store result in ST(0).
DC /4	FSUB <i>m64fp</i>	Valid	Valid	Subtract <i>m64fp</i> 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 <i>m32int</i>	Valid	Valid	Subtract <i>m32int</i> from ST(0) and store result in ST(0).
DE /4	FISUB <i>m16int</i>	Valid	Valid	Subtract <i>m16int</i> 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 3-48 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  $-\infty$  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  $\infty$ , the result is  $\infty$  of the expected sign. If both operands are  $\infty$  of the same sign, an invalid-operation exception is generated.

Table 3-48. 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 – Convert.ToDoubleExtendedPrecisionFP(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.

CO, 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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 <i>m32fp</i>	Valid	Valid	Subtract ST(0) from <i>m32fp</i> and store result in ST(0).
DC /5	FSUBR <i>m64fp</i>	Valid	Valid	Subtract ST(0) from <i>m64fp</i> 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 EO+i	FSUBR ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(i).
DE EO+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 <i>m32int</i>	Valid	Valid	Subtract ST(0) from <i>m32int</i> and store result in ST(0).
DE /5	FISUBR <i>m16int</i>	Valid	Valid	Subtract ST(0) from <i>m16int</i> 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  $\infty$ , the result is  $\infty$  of the expected sign. If both operands are  $\infty$  of the same sign, an invalid-operation exception is generated.

Table 3-49. 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 ← Convert.ToDoubleExtendedPrecisionFP(SRC) – DEST;
ELSE (* Source operand is floating-point value *)
    DEST ← SRC – DEST; Fl;
```

IF Instruction = FSUBRP

```

THEN
    PopRegisterStack; Fl;
```

**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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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	CRO.EM[bit 2] or CRO.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 3-50. 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 ModR/M” 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  $(-0.0 \leftarrow +0.0)$ .

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  $\leftarrow 111$ ;  
 ST(0) > 0.0: C3, C2, C0  $\leftarrow 000$ ;  
 ST(0) < 0.0: C3, C2, C0  $\leftarrow 001$ ;  
 ST(0) = 0.0: C3, C2, C0  $\leftarrow 100$ ;  
 ESAC;

### FPU Flags Affected

C1 Set to 0.  
 C0, C2, C3 See Table 3-50.

### 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 CRO.EM[bit 2] or CRO.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 3-51. 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 ModR/M” 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;
  Fl;
Fl;

IF Instruction = FUCOMP
  THEN
    PopRegisterStack;
Fl;

IF Instruction = FUCOMPP
  THEN
    PopRegisterStack;
Fl;

```

**FPU Flags Affected**

C1                    Set to 0 if stack underflow occurred.  
 C0, C2, C3        See Table 3-51.

**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                CRO.EM[bit 2] or CRO.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 ModR/M

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 3-52. 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

$C1 \leftarrow \text{sign bit of ST}; (* 0 for positive, 1 for negative *)$

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 3-52.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM                  CRO.EM[bit 2] or CRO.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 if stack underflow occurred; otherwise, set to 1.

C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

### Protected Mode Exceptions

#NM CRO.EM[bit 2] or CRO.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
OF AE /1	M	Valid	Valid	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .
FXRSTOR <i>m512byte</i>				
REX.W+ OF AE /1	M	Valid	N.E.	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .
FXRSTOR64 <i>m512byte</i>				

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### 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 3-53 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 3-56 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-57 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 3-53, Table 3-56, or Table 3-57. 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

(x87 FPU, MMX, XMM7-XMM0, MXCSR) ← Load(SRC);

### 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 CRO.TS[bit 3] = 1.
	If CRO.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 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 CRO.TS[bit 3] = 1.
	If CRO.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 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 CRO.TS[bit 3] = 1.
	If CRO.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 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
OF AE /0	M	Valid	Valid	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m</i> 512byte.
FXSAVE <i>m</i> 512byte				
REX.W+ OF AE /0	M	Valid	N.E.	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m</i> 512byte.
FXSAVE64 <i>m</i> 512byte				

### Instruction Operand Encoding

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

### 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 3-53 shows the layout of the state information in memory when the processor is operating in legacy modes.

**Table 3-53. 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	FPU CS							FOP	Rsvd	FTW		FSW		FCW	0	
MXCSR_MASK									Rsvd		FPU DS			FPU DP	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	

**Table 3-53. 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															288	
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 3-53 are defined in Table 3-54.

**Table 3-54. Field Definitions**

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , 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 <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for the layout of the x87 FPU opcode field.
FPU IP	x87 FPU Instruction Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-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. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for a description of the x87 FPU instruction pointer.
FPU CS	x87 FPU Instruction Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates the FPU CS and FPU DS values, and this field is saved as 0000H.

**Table 3-54. Field Definitions (Contd.)**

Field	Definition
FPU DP	x87 FPU Instruction Operand (Data) Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-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. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for a description of the x87 FPU operand pointer.
FPU DS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates the FPU CS and FPU DS values, and this field is saved as 0000H.
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , 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 <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , 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 FP data (assuming the stored data was not the contents of MMX technology registers) using Table 3-55.

**Table 3-55. 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 3-56), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-57), the FPU IP and FPU DP pointers are 32-bits.

**Table 3-56. 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								
FPU IP				FOP	Reserved	FTW	FSW	FCW								0							
MXCSR_MASK		MXCSR				FPU DP										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							

**Table 3-56. Layout of the 64-bit-mode FXSAVE64 Map  
(requires REX.W = 1) (Contd.)**

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0		
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 3-57. 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	FPU CS	FPU IP		FOP	Reserved	FTW	FSW	FCW	0						
MXCSR_MASK		MXCSR			Reserved	FPU DS		FPU DP		16					
Reserved										32					
Reserved										48					
Reserved										64					
Reserved										80					
Reserved										96					
Reserved										112					
Reserved										128					
Reserved										144					
XMM0										160					

**Table 3-57. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0) (Contd.) (Contd.)**

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
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, XMM7-XMM0,
                                         MXCSR);
      ELSE
        DEST ← Save64BitDefaultFxsave(x87 FPU, MMX, XMM7-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 CRO.TS[bit 3] = 1. If CRO.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 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 CRO.TS[bit 3] = 1. If CRO.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 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 CRO.TS[bit 3] = 1. If CRO.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 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  $\log_2(x)$  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 $\pm 0$ .
#D	Source operand is a denormal value.

### Protected Mode Exceptions

#NM	CRO.EM[bit 2] or CRO.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 3-58. FYL2X Results**

		ST(0)							
		$-\infty$	$-F$	$\pm 0$	$+0 < +F < +1$	$+1$	$+F > +1$	$+\infty$	NaN
ST(1)	$-\infty$	*	*	$+\infty$	$+\infty$	*	$-\infty$	$-\infty$	NaN
	$-F$	*	*	**	$+F$	$-0$	$-F$	$-\infty$	NaN
	$-0$	*	*	*	$+0$	$-0$	$-0$	*	NaN
	$+0$	*	*	*	$-0$	$+0$	$+0$	*	NaN
	$+F$	*	*	**	$-F$	$+0$	$+F$	$+\infty$	NaN
	$+\infty$	*	*	$-\infty$	$-\infty$	*	$+\infty$	$+\infty$	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  $\pm 0$ , the instruction returns  $\infty$  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):

$$\log_b x \leftarrow (\log_2 b)^{-1} * \log_2 x$$

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

**Operation**

$ST(1) \leftarrow ST(1) * \log_2 ST(0);$   
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	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	CRO.EM[bit 2] or CRO.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  $-\infty$  to  $+\infty$ . 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 3-59. FYL2XP1 Results

		ST(0)				
		-(1 - ( $\sqrt{2}/2$ )) to -0	-0	+0	+0 to +(1 - ( $\sqrt{2}/2$ ))	NaN
ST(1)	- $\infty$	+ $\infty$	*	*	- $\infty$	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
	+ $\infty$	- $\infty$	*	*	+ $\infty$	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 ( $\epsilon$ ) values, more significant digits can be retained by using the FYL2XP1 instruction than by using  $(\epsilon+1)$  as an argument to the FYL2X instruction. The  $(\epsilon+1)$  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:

$$\text{scale factor} \leftarrow \log_n 2$$

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

### Operation

$ST(1) \leftarrow ST(1) * \log_2(ST(0) + 1.0);$   
 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	CRO.EM[bit 2] or CRO.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.

**HADDPD—Packed Double-FP Horizontal Add**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7C /r HADDPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Horizontal add packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 7C /r VHADDPD <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Horizontal add packed double-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 7C /r VHADDPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Horizontal add packed double-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

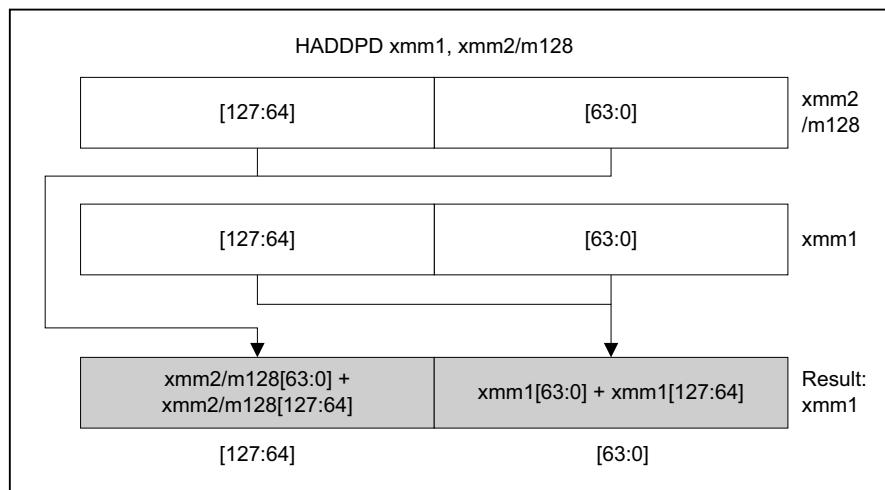
**Description**

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.

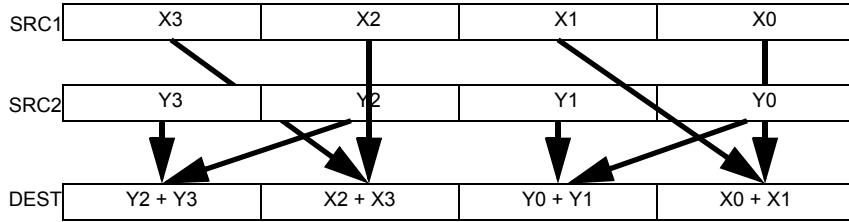
In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-15 for HADDPD; see Figure 3-16 for VHADDPD.



OM15993

**Figure 3-15. HADDPD—Packed Double-FP Horizontal Add**

**Figure 3-16. 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

## Operation

### 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[VLMAX-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[VLMAX-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 Exceptions Type 2.

## HADDPS—Packed Single-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 7C /r HADDPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Horizontal add packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG 7C /r VHADDPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Horizontal add packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.F2.0F.WIG 7C /r VHADDPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Horizontal add packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

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

### 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.

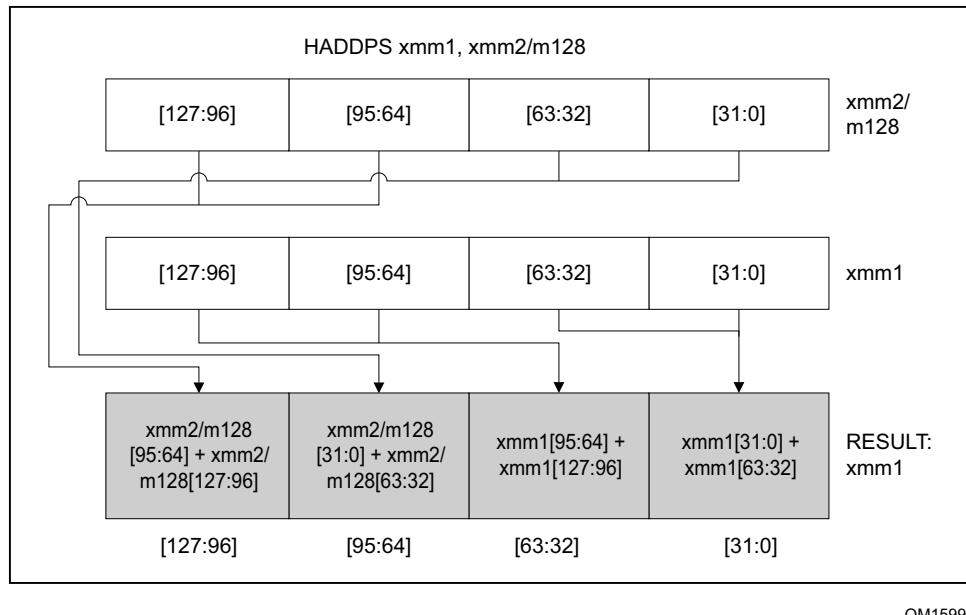
Adds single-precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

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.

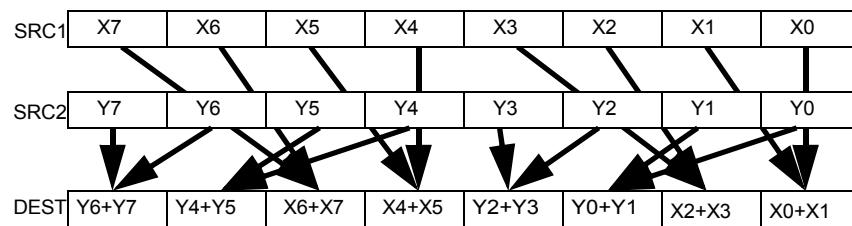
In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-17 for HADDPS; see Figure 3-18 for VHADDPS.



OM15994

**Figure 3-17. HADDPS—Packed Single-FP Horizontal Add**



**Figure 3-18. 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

## Operation

### 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[VLMAX-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[VLMAX-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 Exceptions Type 2.

**HLT—Halt**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F4	HLT	NP	Valid	Valid	Halt

**Instruction Operand Encoding**

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

**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.

## HSUBPD—Packed Double-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7D /r HSUBPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Horizontal subtract packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 7D /r VHSUBPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Horizontal subtract packed double-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 7D /r VHSUBPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Horizontal subtract packed double-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

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

### Description

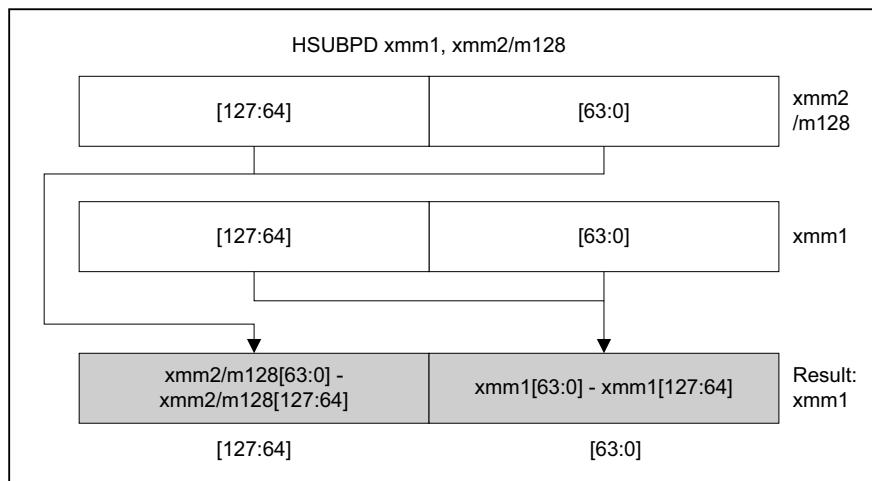
The HSUBPD instruction subtracts horizontally the packed DP FP numbers of both operands.

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.

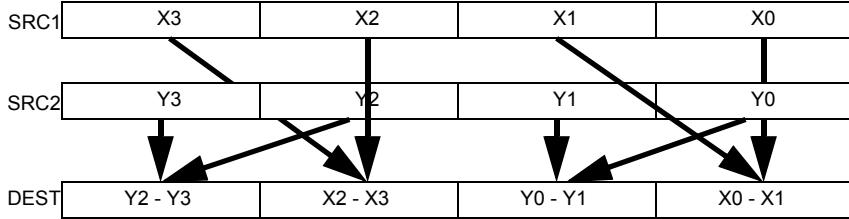
In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-19 for HSUBPD; see Figure 3-20 for VHSUBPD.



OM15995

Figure 3-19. HSUBPD—Packed Double-FP Horizontal Subtract

**Figure 3-20. 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

## Operation

### 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[VLMAX-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[VLMAX-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 Exceptions Type 2.

**HSUBPS—Packed Single-FP Horizontal Subtract**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 7D /r HSUBPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Horizontal subtract packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG 7D /r VHSUBPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Horizontal subtract packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.F2.0F.WIG 7D /r VHSUBPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Horizontal subtract packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

**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.

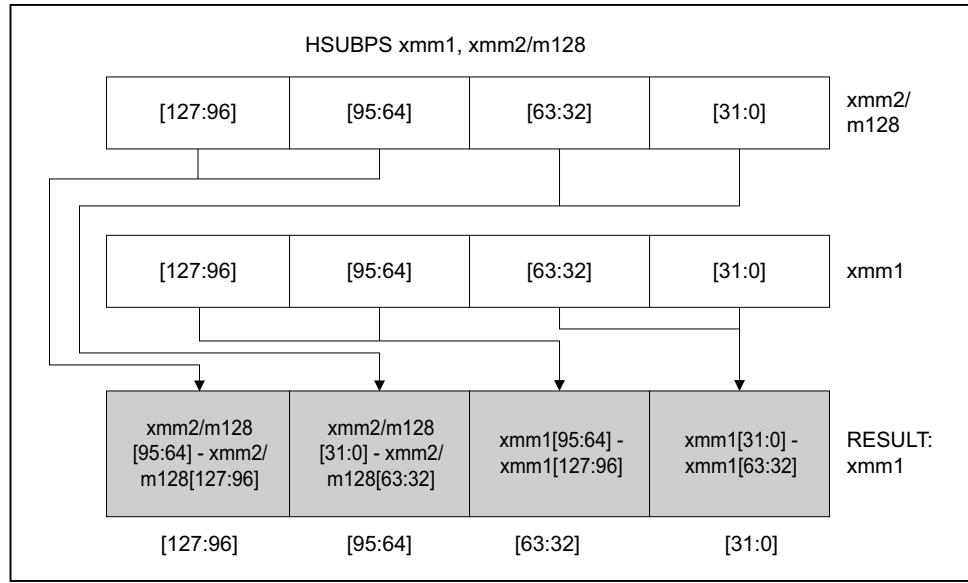
Subtracts the single-precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

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.

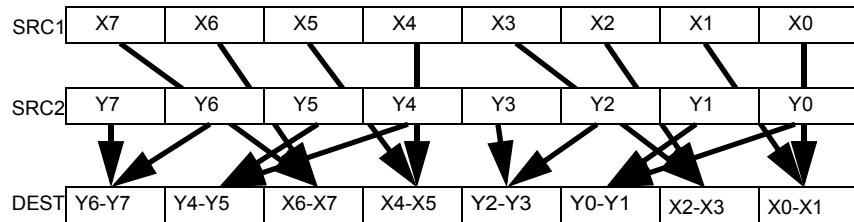
Subtracts the single-precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

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

See Figure 3-21 for HSUBPS; see Figure 3-22 for VHSUBPS.



OM15996

**Figure 3-21. HSUBPS—Packed Single-FP Horizontal Subtract****Figure 3-22. 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

## Operation

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

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$   
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

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

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

### **VHSUBPS (VEX.256 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC1}[191:160]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[223:192] - \text{SRC1}[255:224]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC2}[159:128] - \text{SRC2}[191:160]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[223:192] - \text{SRC2}[255:224]$

## 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 Exceptions Type 2.

## IDIV—Signed Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /7	IDIV r/m8	M	Valid	Valid	Signed divide AX by r/m8, with result stored in: AL ← Quotient, AH ← Remainder.
REX + F6 /7	IDIV r/m8*	M	Valid	N.E.	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:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

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 3-60.

**Table 3-60. 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 $2^{32} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	- $2^{63}$ to $2^{64} - 1$

## 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 > 7FFFFH) or (temp < 80000H)
        (* If a positive result is greater than 7FFFFH
        or a negative result is less than 80000H *)
        THEN
            #DE; (* Divide error *)
        ELSE
            AX ← temp;
            DX ← DX:AX SignedModulus SRC;
        FI;
    FI;
ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
temp ← EDX:EAX / SRC; (* Signed division *)
IF (temp > 7FFFFFFFH) or (temp < 800000000H)
    (* If a positive result is greater than 7FFFFFFFH
    or a negative result is less than 800000000H *)
    THEN
        #DE; (* Divide error *)
    ELSE
        EAX ← temp;
        EDX ← EDX:EAX SignedModulus SRC;
    FI;
FI;
ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
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;

```

## 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*	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.
OF AF /r	IMUL r16, r/m16	RM	Valid	Valid	word register ← word register * r/m16.
OF AF /r	IMUL r32, r/m32	RM	Valid	Valid	doubleword register ← doubleword register * r/m32.
REX.W + OF 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:**

- \* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	NA

**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 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 product is then 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 product is then 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 significant bit (including the sign bit) are carried into the upper half of the result. The CF and OF flags are cleared when the result (including the sign bit) fits exactly in the lower half of the result.

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
      AX ← AL * SRC (* Signed multiplication *)
      IF AL = AX
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; Fl;
    ELSE IF OperandSize = 16
      THEN
        DX:AX ← AX * SRC (* Signed multiplication *)
        IF sign_extend_to_32 (AX) = DX:AX
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; Fl;
    ELSE IF OperandSize = 32
      THEN
        EDX:EAX ← EAX * SRC (* Signed multiplication *)
        IF EAX = EDX:EAX
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; Fl;
    ELSE (* OperandSize = 64 *)
      RDX:RAX ← RAX * SRC (* Signed multiplication *)
      IF RAX = RDX:RAX
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; Fl;
    Fl;
  Fl;
ELSE IF (NumberOfOperands = 2)
  THEN
    temp ← DEST * SRC (* Signed multiplication; temp is double DEST size *)
  
```

```

DEST ← DEST * SRC (* Signed multiplication *)
IF temp ≠ DEST
    THEN CF ← 1; OF ← 1;
    ELSE CF ← 0; OF ← 0; Fl;
ELSE (* NumberOfOperands = 3 *)
    DEST ← SRC1 * SRC2 (* Signed multiplication *)
    temp ← SRC1 * SRC2 (* Signed multiplication; temp is double SRC1 size *)
    IF temp ≠ DEST
        THEN CF ← 1; OF ← 1;
        ELSE CF ← 0; OF ← 0; Fl;
    Fl;
Fl;

```

### 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 <i>ib</i>	IN AL, <i>imm8</i>	I	Valid	Valid	Input byte from <i>imm8</i> I/O port address into AL.
E5 <i>ib</i>	IN AX, <i>imm8</i>	I	Valid	Valid	Input word from <i>imm8</i> I/O port address into AX.
E5 <i>ib</i>	IN EAX, <i>imm8</i>	I	Valid	Valid	Input dword from <i>imm8</i> I/O port address into EAX.
EC	IN AL,DX	NP	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX,DX	NP	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX,DX	NP	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	NA	NA	NA
NP	NA	NA	NA	NA

### 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 16, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

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

### 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 *)
    Fl;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Read from selected I/O port *)
Fl;

```

### 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 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.
- #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	M	Valid	Valid	Increment r/m byte by 1.
REX + FE /0	INC r/m8*	M	Valid	N.E.	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**	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:**

\* 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.

\*\* 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)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

**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;

**AFlags 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.

**INS/INSB/INSW/INSD—Input from Port to String**

<b>Opcode</b>	<b>Instruction</b>	<b>Op/ En</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
6C	INS <i>m8</i> , DX	NP	Valid	Valid	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.*
6D	INS <i>m16</i> , DX	NP	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. <sup>1</sup>
6D	INS <i>m32</i> , DX	NP	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. <sup>1</sup>
6C	INSB	NP	Valid	Valid	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI. <sup>1</sup>
6D	INSW	NP	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. <sup>1</sup>
6D	INSD	NP	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. <sup>1</sup>

**NOTES:**

- \* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

**Instruction Operand Encoding**

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

**Description**

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/REPE/REPZ /REPNE/REPNZ—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 16, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, 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.

## 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;
```

Fl;  
Fl;

### 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.

## INSERTPS – Insert Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS <i>xmm1, xmm2/m32, imm8</i>	RMI	V/V	SSE4_1	Insert a single precision floating-point value selected by <i>imm8</i> from <i>xmm2/m32</i> into <i>xmm1</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 21 /r ib VININSERTPS <i>xmm1, xmm2, xmm3/m32, imm8</i>	RVMI	V/V	AVX	Insert a single precision floating point value selected by <i>imm8</i> from <i>xmm3/m32</i> and merge into <i>xmm2</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

(register source form)

Select a single precision floating-point element from second source as indicated by Count\_S bits of the immediate operand and insert it into the first source at the location indicated by the Count\_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

(memory source form)

Load a floating-point element from a 32-bit memory location and insert it into the first source at the location indicated by the Count\_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version. The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

## Operation

### **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[VLMAX-1:128] (Unmodified)

```

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

```

IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
  ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
  0: TMP ← SRC2[31:0]
  1: TMP ← SRC2[63:32]
  2: TMP ← SRC2[95:64]
  3: TMP ← SRC2[127:96]
ESAC;

CASE (COUNT_D) OF
  0: TMP2[31:0] ← TMP
    TMP2[127:32] ← SRC1[127:32]
  1: TMP2[63:32] ← TMP
    TMP2[31:0] ← SRC1[31:0]
    TMP2[127:64] ← SRC1[127:64]

```

```

2: TMP2[95:64] ← TMP
   TMP2[63:0] ← SRC1[63:0]
   TMP2[127:96] ← SRC1[127:96]
3: TMP2[127:96] ← TMP
   TMP2[95:0] ← SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
ELSE DEST[127:96] ← TMP2[127:96]
DEST[VLMAX-1:128] ← 0

```

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

INSERTPS: `__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx);`

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

None

#### **Other Exceptions**

See Exceptions Type 5.

**INT *n*/INTO/INT 3—Call to Interrupt Procedure**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT 3	NP	Valid	Valid	Interrupt 3—trap to debugger.
CD <i>ib</i>	INT <i>imm8</i>	I	Valid	Valid	Interrupt vector number specified by immediate byte.
CE	INTO	NP	Invalid	Valid	Interrupt 4—if overflow flag is 1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
I	imm8	NA	NA	NA

**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 an interrupt vector number from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each interrupt vector number provides an index to a gate descriptor in the IDT. The first 32 interrupt vector numbers are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

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), interrupt vector number 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 INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows:

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

Note that the “normal” 2-byte opcode for INT 3 (CD03) does not have these special features. 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 action of the INT *n* instruction (including the INTO and INT 3 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.

The interrupt vector number 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 3-61. 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				
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.

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.

#### Operation

The following operational description applies not only to the INT n and INTO instructions, but also to 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, EXT is 0; otherwise, EXT is 1.

## INSTRUCTION SET REFERENCE, A-M

```
IF PE = 0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE = 1 *)
    IF (VM = 1 and IOPL < 3 AND INT n)
      THEN
        #GP(0); (* Bit 0 of error code is 0 because INT n *)
      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;
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 *)
    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;
```

```

Fl;
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 *)
Fl;
IF software interrupt (* Generated by INT n, INT 3, or INTO *)
    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*)
Fl;
Fl;
IF gate not present
    THEN #NP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
Fl;
GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
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)); Fl;
            (* 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(TSS selector,0,EXT)); Fl;
            (* idt operand to error_code is 0 because selector is used *)
    IF TSS not present
        THEN #NP(TSS selector,0,EXT)); Fl;
            (* 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); Fl;
            Push(error code);
Fl;
IF EIP not within code segment limit
    THEN #GP(EXT); Fl;
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); Fl; (* 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)); Fl;
            (* 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)); Fl;
            (* 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)); Fl;
(* 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; Fl;
      (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
    Fl;
  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 *)
    Fl;
  Fl;
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)); Fl;
      (* 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)); Fl;
      (* 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);
  Fl;
  IF NewSS is NULL
    THEN #TS(EXT); Fl;
  IF NewSS index is not within its descriptor-table limits
  or NewSS RPL ≠ new code-segment DPL

```

```

        THEN #TS(error_code(NewSS,0,EXT)); Fl;
        (* 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)); Fl;
        (* idt operand to error_code is 0 because selector is used *)
    IF NewSS is not present
        THEN #SS(error_code(NewSS,0,EXT)); Fl;
        (* idt operand to error_code is 0 because selector is used *)
    ELSE (* IA-32e mode *)
        IF IDT-gate IST = 0
            THEN TSSstackAddress ← (new code-segment DPL « 3) + 4;
            ELSE TSSstackAddress ← (IDT gate IST « 3) + 28;
        Fl;
        IF (TSSstackAddress + 7) > current TSS limit
            THEN #TS(error_code(current TSS selector,0,EXT)); Fl;
            (* 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 *)
    Fl;
    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)); Fl;
                (* idt operand to error_code is 0 because selector is used *)
        Fl
    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)); Fl;
                    (* idt operand to error_code is 0 because selector is used *)
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(EXT); Fl; (* Error code contains NULL selector *)
            Fl;
    Fl;
    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); Fl; (* 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); Fl; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFFFFFFFFOH;
            SS ← NewSS;
        Fl;
    IF IDT gate is 32-bit
        THEN

```

## INSTRUCTION SET REFERENCE, A-M

```
    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;
            CPL ← new code-segment DPL;
            CS(RPL) ← CPL;
            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)); Fl;
    (* 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);
Fl;
IF NewSS is NULL
  THEN #TS(EXT); Fl; (* 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)); Fl;
  (* 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)); Fl;
  (* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
  THEN #SS(error_code(NewSS,0,EXT)); Fl;
  (* idt operand to error_code is 0 because selector is used *)
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)); Fl;
      (* 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)); Fl;
        (* idt operand to error_code is 0 because selector is used *)
Fl;
IF instruction pointer from IDT gate is not within new code-segment limits
  THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
NT ← 0;
IF service through interrupt gate
  THEN IF = 0; Fl;
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);

```

## INSTRUCTION SET REFERENCE, A-M

```
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:IP ← Gate(CS); (* Segment descriptor information also loaded *)
IF OperandSize = 32
    THEN
        EIP ← Gate(instruction pointer);
    ELSE (* OperandSize is 16 *)
        EIP ← Gate(instruction pointer) AND 0000FFFFH;
    FI;
    (* Start execution of new routine in Protected Mode *)
END;
INTRAPRIVILEGELEVELINTERRUPT:
(* 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);
        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 & FFFFFFFFFFFFFFOH;
                        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 *)
                        FI;
```

```

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 IDT gate is interrupt gate
  THEN IF ← 0; Fl; (* 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 interrupt vector number is outside the IDT limits.  
                       If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.  
                       If an interrupt is generated by the INT *n*, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor 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.
- #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-, trap-, or 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 <i>n</i> , INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor 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-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits. If the interrupt vector number is outside the IDT limits. If an IDT descriptor is not an interrupt-, trap-, or task-descriptor. If an interrupt is generated by the INT <i>n</i> instruction and the DPL of an interrupt-, trap-, or task-descriptor 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-, trap-, or 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.

#BP	If the INT 3 instruction is executed.
#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

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(error_code)	If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical. If the segment selector in the 64-bit interrupt or trap gate is NULL. If the interrupt vector number is outside the IDT limits. If the interrupt vector number points to a gate which is in non-canonical space. If the interrupt vector number points to a descriptor which is not a 64-bit interrupt gate or 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.
#SS(error_code)	If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch. If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).
#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.
#PF(fault-code)	If the RSP from the TSS is outside descriptor table limits. 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.

**INVD—Invalidate Internal Caches**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 08	INVD	NP	Valid	Valid	Flush internal caches; initiate flushing of external caches.

**NOTES:**

- \* See the IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

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

**Description**

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.

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. |
| #UD    | If the LOCK prefix is used.              |

**Real-Address Mode Exceptions**

#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 Entry**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01/7	INVLPG <i>m</i>	M	Valid	Valid	Invalidate TLB Entry for page that contains <i>m</i> .

**NOTES:**

- \* 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)	NA	NA	NA

**Description**

Invalidates (flushes) the translation lookaside buffer (TLB) entry specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes the TLB entry for that page.

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 the TLB entry only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidates only TLB entries associated with the current PCID. (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” and Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” of 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**

Flush(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.

**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**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32- bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
66 0F 38 82 /r INVPCID r32, m128		RM	NE/V	INVPCID Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r32 and descriptor in m128.
66 0F 38 82 /r INVPCID r64, m128		RM	V/NE	INVPCID Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r64 and descriptor in m128.

**Instruction Operand Encoding**

<b>Op/En</b>	<b>Operand 1</b>	<b>Operand 2</b>	<b>Operand 3</b>	<b>Operand 4</b>
RM	ModRM:reg (R)	ModRM:r/m (R)	NA	NA

**Description**

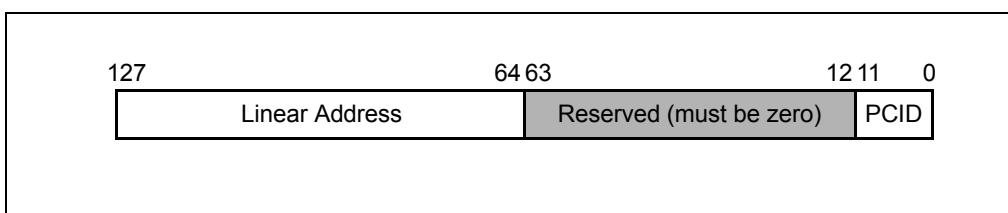
Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on process-context identifier (PCID). (See Section 4.10, “Caching Translation Information,” in *Intel 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A*.) Invalidation is based on the INVPCID 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. 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 3-23. 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 3-23. INVPCID Descriptor**

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 64-bit mode (see Chapter 4.10.1, “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.(EAX=07H, ECX=0H):EBX.INVPCID (bit 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.(EAX=07H, ECX=0H):EBX.INVPCID (bit 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.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.

## IRET/IRED—Interrupt Return

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CF	IRET	NP	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRED	NP	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	NP	Valid	N.E.	Interrupt return (64-bit operand size).

### Instruction Operand Encoding

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

### 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 7 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

IRET and IRED are mnemonics for the same opcode. The IRED 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.

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.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

```

IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE
        IF (IA32_EFER.LMA = 0)
            THEN (* Protected mode *)
                GOTO PROTECTED-MODE;
            ELSE (* IA-32e mode *)
                GOTO IA-32e-MODE;
        FI;
    FI;
REAL-ADDRESS-MODE;
IF OperandSize = 32
    THEN
        IF top 12 bytes of stack not within stack limits
            THEN #SS; FI;
        tempEIP ← 4 bytes at end of stack
        IF tempEIP[31:16] is not zero THEN #GP(0); FI;
        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 *)
        IF top 6 bytes of stack are not within stack limits
            THEN #SS; FI;
        EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
        CS ← Pop(); (* 16-bit pop *)
        EFLAGS[15:0] ← Pop();
    FI;
END;
PROTECTED-MODE:
IF VM = 1 (* Virtual-8086 mode: PE = 1, VM = 1 *)
    THEN
        GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE = 1, VM = 1 *)
FI;
IF NT = 1
    THEN
        GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
FI;
IF OperandSize = 32
    THEN
        IF top 12 bytes of stack not within stack limits
            THEN #SS(0); FI;
        tempEIP ← Pop();
        tempCS ← Pop();
        tempEFLAGS ← Pop();
    ELSE (* OperandSize = 16 *)
        IF top 6 bytes of stack are not within stack limits

```

```

        THEN #SS(0); Fl;
tempEIP ← Pop();
tempCS ← Pop();
tempEFLAGS ← Pop();
tempEIP ← tempEIP AND FFFFH;
tempEFLAGS ← tempEFLAGS AND FFFFH;

Fl;
IF tempEFLAGS(VM) = 1 and CPL = 0
    THEN
        GOTO RETURN-TO-VIRTUAL-8086-MODE;
    ELSE
        GOTO PROTECTED-MODE-RETURN;
Fl;
IA-32e-MODE:
IF NT = 1
    THEN #GP(0);
ELSE IF OperandSize = 32
    THEN
        IF top 12 bytes of stack not within stack limits
            THEN #SS(0); Fl;
        tempEIP ← Pop();
        tempCS ← Pop();
        tempEFLAGS ← Pop();
    ELSE IF OperandSize = 16
        THEN
            IF top 6 bytes of stack are not within stack limits
                THEN #SS(0); Fl;
            tempEIP ← Pop();
            tempCS ← Pop();
            tempEFLAGS ← Pop();
            tempEIP ← tempEIP AND FFFFH;
            tempEFLAGS ← tempEFLAGS AND FFFFH;

        Fl;
    ELSE (* OperandSize = 64 *)
        THEN
            tempRIP ← Pop();
            tempCS ← Pop();
            tempEFLAGS ← Pop();
            tempRSP ← Pop();
            tempSS ← Pop();

        Fl;
    GOTO IA-32e-MODE-RETURN;

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
            IF top 12 bytes of stack not within stack limits
                THEN #SS(0); Fl;
            IF instruction pointer not within code segment limits
                THEN #GP(0); Fl;
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)

```

## INSTRUCTION SET REFERENCE, A-M

```
EFLAGS ← Pop();
(* VM, IOPL,VIP and VIF EFLAG bits not modified by pop *)
ELSE (* OperandSize = 16 *)
    IF top 6 bytes of stack are not within stack limits
        THEN #SS(0); Fl;
    IF instruction pointer not within code segment limits
        THEN #GP(0); Fl;
    EIP ← Pop();
    EIP ← EIP AND 0000FFFFH;
    CS ← Pop(); (* 16-bit pop *)
    EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
Fl;
ELSE
    #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
Fl;
END;
```

### RETURN-TO-VIRTUAL-8086-MODE:

```
(* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
IF top 24 bytes of stack are not within stack segment limits
    THEN #SS(0); Fl;
IF instruction pointer not within code segment limits
    THEN #GP(0); Fl;
CS ← tempCS;
EIP ← tempEIP & FFFFH;
EFLAGS ← tempEFLAGS;
TempESP ← Pop();
TempSS ← Pop();
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 *)
SS:ESP ← TempSS:TempESP;
CPL ← 3;
(* Resume execution in Virtual-8086 mode *)
END;
```

### TASK-RETURN: (\* PE = 1, VM = 0, NT = 1 \*)

```
Read segment selector in link field of current TSS;
IF local/global bit is set to local
or index not within GDT limits
    THEN #TS (TSS selector); Fl;
Access TSS for task specified in link field of current TSS;
IF TSS descriptor type is not TSS or if the TSS is marked not busy
    THEN #TS (TSS selector); Fl;
IF TSS not present
    THEN #NP(TSS selector); Fl;
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 code segment limit
    THEN #GP(0); Fl;
END;
```

### PROTECTED-MODE-RETURN: (\* PE = 1 \*)

```

IF return code segment selector is NULL
    THEN GP(0); Fl;
IF return code segment selector addresses descriptor beyond descriptor table limit
    THEN GP(selector); Fl;
Read segment descriptor pointed to by the return code segment selector;
IF return code segment descriptor is not a code segment
    THEN #GP(selector); Fl;
IF return code segment selector RPL < CPL
    THEN #GP(selector); Fl;
IF return code segment descriptor is conforming
and return code segment DPL > return code segment selector RPL
    THEN #GP(selector); Fl;
IF return code segment descriptor is not present
    THEN #NP(selector); Fl;
IF return code segment selector RPL > CPL
    THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; Fl;
END;

```

RETURN-TO-SAME-PRIVILEGE-LEVEL: (\* PE = 1, RPL = CPL \*)

```

IF new mode ≠ 64-Bit Mode
    THEN
        IF tempEIP is not within code segment limits
            THEN #GP(0); Fl;
        EIP ← tempEIP;
        ELSE (* new mode = 64-bit mode *)
            IF tempRIP is non-canonical
                THEN #GP(0); Fl;
            RIP ← tempRIP;
    Fl;
    CS ← tempCS; (* Segment descriptor information also loaded *)
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize = 32 or OperandSize = 64
        THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; Fl;
    IF CPL ≤ IOPL
        THEN EFLAGS(IF) ← tempEFLAGS; Fl;
    IF CPL = 0
        THEN (* VM = 0 in flags image *)
            EFLAGS(IOPL) ← tempEFLAGS;
        IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(VIF, VIP) ← tempEFLAGS; Fl;
    Fl;
END;

```

RETURN-TO-OUTER-PRIVILEGE-LEVEL:

```

IF OperandSize = 32
    THEN
        IF top 8 bytes on stack are not within limits
            THEN #SS(0); Fl;
        ELSE (* OperandSize = 16 *)
            IF top 4 bytes on stack are not within limits
                THEN #SS(0); Fl;
    Fl;
    Read return segment selector;

```

## INSTRUCTION SET REFERENCE, A-M

IF stack segment selector is NULL  
    THEN #GP(0); Fl;

IF return stack segment selector index is not within its descriptor table limits  
    THEN #GP(SSselector); Fl;

Read segment descriptor pointed to by return segment selector;  
IF stack segment selector RPL ≠ RPL of the return code segment selector  
or the stack segment descriptor does not indicate a a writable data segment;  
or the stack segment DPL ≠ RPL of the return code segment selector  
    THEN #GP(SS selector); Fl;

IF stack segment is not present  
    THEN #SS(SS selector); Fl;

IF new mode ≠ 64-Bit Mode  
    THEN  
        IF tempEIP is not within code segment limits  
            THEN #GP(0); Fl;  
            EIP ← tempEIP;  
        ELSE (\* new mode = 64-bit mode \*)  
            IF tempRIP is non-canonical  
                THEN #GP(0); Fl;  
                RIP ← tempRIP;  
        Fl;  
        CS ← tempCS;  
        EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;  
        IF OperandSize = 32  
            THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; Fl;  
        IF CPL ≤ IOPL  
            THEN EFLAGS(IF) ← tempEFLAGS; Fl;  
        IF CPL = 0  
            THEN  
                EFLAGS(IOPL) ← tempEFLAGS;  
                IF OperandSize = 32  
                    THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS; Fl;  
                IF OperandSize = 64  
                    THEN EFLAGS(VIF, VIP) ← tempEFLAGS; Fl;  
        Fl;  
        CPL ← RPL of the return code segment selector;  
        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 (\* Stored in hidden part of segment register \*)  
                    THEN (\* Segment register invalid \*)  
                        SegmentSelector ← 0; (\* NULL segment selector \*)  
                Fl;  
            OD;  
    END;

IA-32e-MODE-RETURN: (\* IA32\_EFER.LMA = 1, PE = 1 \*)  
IF ( (return code segment selector is NULL) or (return RIP is non-canonical) or  
    (SS selector is NULL going back to compatibility mode) or  
    (SS selector is NULL going back to CPL3 64-bit mode) or  
    (RPL <> CPL going back to non-CPL3 64-bit mode for a NULL SS selector) )  
    THEN GP(0); Fl;

IF return code segment selector addresses descriptor beyond descriptor table limit  
    THEN GP(selector); Fl;

```

Read segment descriptor pointed to by the return code segment selector;
IF return code segment descriptor is not a code segment
    THEN #GP(selector); Fl;
IF return code segment selector RPL < CPL
    THEN #GP(selector); Fl;
IF return code segment descriptor is conforming
and return code segment DPL > return code segment selector RPL
    THEN #GP(selector); Fl;
IF return code segment descriptor is not present
    THEN #NP(selector); Fl;
IF return code segment selector RPL > CPL
    THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; Fl;
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.
#NP(selector)	If the return code or stack 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.

### 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 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.
#SS(0)	If the stack segment selector RPL is not equal to the RPL of the return code segment selector. 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 an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

## 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	JAE rel32	D	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 cw	JA rel16	D	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 83 cd	JAE rel32	D	Valid	Valid	Jump near if above or equal (CF=0).
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).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 85 cw	JNE <i>rel16</i>	D	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.
OF 85 cd	JNE <i>rel32</i>	D	Valid	Valid	Jump near if not equal (ZF=0).
OF 8E cw	JNG <i>rel16</i>	D	N.S.	Valid	Jump near if not greater (ZF=1 or SF ≠ OF). Not supported in 64-bit mode.
OF 8E cd	JNG <i>rel32</i>	D	Valid	Valid	Jump near if not greater (ZF=1 or SF ≠ OF).
OF 8C cw	JNGE <i>rel16</i>	D	N.S.	Valid	Jump near if not greater or equal (SF ≠ OF). Not supported in 64-bit mode.
OF 8C cd	JNGE <i>rel32</i>	D	Valid	Valid	Jump near if not greater or equal (SF ≠ OF).
OF 8D cw	JNL <i>rel16</i>	D	N.S.	Valid	Jump near if not less (SF=OF). Not supported in 64-bit mode.
OF 8D cd	JNL <i>rel32</i>	D	Valid	Valid	Jump near if not less (SF=OF).
OF 8F cw	JNLE <i>rel16</i>	D	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.
OF 8F cd	JNLE <i>rel32</i>	D	Valid	Valid	Jump near if not less or equal (ZF=0 and SF=OF).
OF 81 cw	JNO <i>rel16</i>	D	N.S.	Valid	Jump near if not overflow (OF=0). Not supported in 64-bit mode.
OF 81 cd	JNO <i>rel32</i>	D	Valid	Valid	Jump near if not overflow (OF=0).
OF 8B cw	JNP <i>rel16</i>	D	N.S.	Valid	Jump near if not parity (PF=0). Not supported in 64-bit mode.
OF 8B cd	JNP <i>rel32</i>	D	Valid	Valid	Jump near if not parity (PF=0).
OF 89 cw	JNS <i>rel16</i>	D	N.S.	Valid	Jump near if not sign (SF=0). Not supported in 64-bit mode.
OF 89 cd	JNS <i>rel32</i>	D	Valid	Valid	Jump near if not sign (SF=0).
OF 85 cw	JNZ <i>rel16</i>	D	N.S.	Valid	Jump near if not zero (ZF=0). Not supported in 64-bit mode.
OF 85 cd	JNZ <i>rel32</i>	D	Valid	Valid	Jump near if not zero (ZF=0).
OF 80 cw	JO <i>rel16</i>	D	N.S.	Valid	Jump near if overflow (OF=1). Not supported in 64-bit mode.
OF 80 cd	JO <i>rel32</i>	D	Valid	Valid	Jump near if overflow (OF=1).
OF 8A cw	JP <i>rel16</i>	D	N.S.	Valid	Jump near if parity (PF=1). Not supported in 64-bit mode.
OF 8A cd	JP <i>rel32</i>	D	Valid	Valid	Jump near if parity (PF=1).
OF 8A cw	JPE <i>rel16</i>	D	N.S.	Valid	Jump near if parity even (PF=1). Not supported in 64-bit mode.
OF 8A cd	JPE <i>rel32</i>	D	Valid	Valid	Jump near if parity even (PF=1).
OF 8B cw	JPO <i>rel16</i>	D	N.S.	Valid	Jump near if parity odd (PF=0). Not supported in 64-bit mode.
OF 8B cd	JPO <i>rel32</i>	D	Valid	Valid	Jump near if parity odd (PF=0).
OF 88 cw	JS <i>rel16</i>	D	N.S.	Valid	Jump near if sign (SF=1). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
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	NA	NA	NA

### 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 Jcc 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 Jcc 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 Jcc 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 Jcc 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 Jcc 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;

```

## 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	D	Inv.	Valid	Jump far, absolute, address given in operand
EA cp	JMP ptr16:32	D	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP m16:16	D	Valid	Valid	Jump far, absolute indirect, address given in m16:16
FF /5	JMP m16:32	D	Valid	Valid	Jump far, absolute indirect, address given in m16:32.
REX.W + FF /5	JMP m16:64	D	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
		NA	NA		
D	Offset	NA	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA	NA

**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 7, 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 7 in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for detailed information on the mechanics of a task switch.

Note that when you execute a task switch with a JMP instruction, 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.

**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.

See the summary chart at the beginning of this section for encoding data and limits.

## 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;
  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;
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 (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
            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 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;
    CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL
    EIP ← tempEIP;
END;

```

## INSTRUCTION SET REFERENCE, A-M

### NONCONFORMING-CODE-SEGMENT:

```
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); Fl;
IF (RPL > CPL) OR (DPL ≠ CPL)
    THEN #GP(code segment selector); Fl;
IF segment not present
    THEN #NP(segment selector); Fl;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; Fl;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
and tempEIP outside code segment limit
    THEN #GP(0); Fl
IF tempEIP is non-canonical THEN #GP(0); Fl;
CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
END;
```

### CALL-GATE:

```
IF call gate DPL < CPL
or call gate DPL < call gate segment-selector RPL
    THEN #GP(call gate selector); Fl;
IF call gate not present
    THEN #NP(call gate selector); Fl;
IF call gate code-segment selector is NULL
    THEN #GP(0); Fl;
IF call gate code-segment selector index outside descriptor table limits
    THEN #GP(code segment selector); Fl;
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); Fl;
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); Fl;
```

```
IF code segment is not present
    THEN #NP(code-segment selector); Fl;
IF instruction pointer is not within code-segment limit
    THEN #GP(0); Fl;
tempEIP ← DEST(Offset);
IF GateSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; Fl;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
outside code segment limit
    THEN #GP(0); Fl
CS ← DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
```

```
END;
```

### TASK-GATE:

```
IF task gate DPL < CPL
or task gate DPL < task gate segment-selector RPL
```

```

THEN #GP(task gate selector); Fl;
IF task gate not present
    THEN #NP(gate selector); Fl;
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 TSS descriptor specifies that the TSS is busy
    THEN #GP(TSS selector); Fl;
IF TSS not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
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); Fl;
IF TSS is not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
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.
#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.

### 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.
#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-confirming and CPL ≠ DPL.
- If the code segment is confirming and CPL < DPL.
- If a code segment or 64-bit call gate is not present.
- (64-bit mode only) If a far jump is direct to an absolute address in memory.
- If the LOCK prefix is used.
- If a page fault occurs.
- If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**LAHF—Load Status Flags into AH Register**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9F	LAHF	NP	Invalid*	Valid	Load: AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF).

**NOTES:**

\*Valid in specific steppings. See Description section.

**Instruction Operand Encoding**

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

**Description**

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

**Operation**

```
IF 64-Bit Mode
  THEN
    IF CPUID.80000001H:ECX.LAHF-SAHF[bit 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[bit 0] = 0.
 If the LOCK prefix is used.

## LAR—Load Access Rights Byte

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 02 /r	LAR <i>r16, r16/m16</i>	RM	Valid	Valid	<i>r16</i> ← access rights referenced by <i>r16/m16</i>
OF 02 /r	LAR <i>reg, r32/m16</i> <sup>1</sup>	RM	Valid	Valid	<i>reg</i> ← access rights referenced by <i>r32/m16</i>

**NOTES:**

1. For all loads (regardless of source or destination sizing) only bits 16-0 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)	NA	NA

### 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 flag 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.

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 3-62.
- 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 3-62. 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	No
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;
  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 OF F0 /r LDDQU <i>xmm1, mem</i>	RM	V/V	SSE3	Load unaligned data from <i>mem</i> and return double quadword in <i>xmm1</i> .
VEX.128.F2.0F.WIG F0 /r VLDDQU <i>xmm1, m128</i>	RM	V/V	AVX	Load unaligned packed integer values from <i>mem</i> to <i>xmm1</i> .
VEX.256.F2.0F.WIG F0 /r VLDDQU <i>ymm1, m256</i>	RM	V/V	AVX	Load unaligned packed integer values from <i>mem</i> to <i>ymm1</i> .

**Instruction Operand Encoding**

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

**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 (CRO.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[VLMAX-1:128] (Unmodified)

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

DEST[127:0]  $\leftarrow$  SRC[127:0]  
DEST[VLMAX-1:128]  $\leftarrow$  0

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

DEST[255:0]  $\leftarrow$  SRC[255:0]

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

LDDQU: `__m128i _mm_lddqu_si128 (__m128i * p);`  
LDDQU: `__m256i _mm256_lddqu_si256 (__m256i * p);`

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4;

Note treatment of #AC varies.

**LDMXCSR—Load MXCSR Register**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32-bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
OF,AE,/2 LDMXCSR <i>m32</i>	M	V/V	SSE	Load MXCSR register from <i>m32</i> .
VEX.LZ.OF.WIG AE /2 VLDMXCSR <i>m32</i>	M	V/V	AVX	Load MXCSR register from <i>m32</i> .

**Instruction Operand Encoding**

<b>Op/En</b>	<b>Operand 1</b>	<b>Operand 2</b>	<b>Operand 3</b>	<b>Operand 4</b>
M	ModRM:r/m (r)	NA	NA	NA

**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**

$\text{MXCSR} \leftarrow \text{m32};$

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

`_mm_setcsr(unsigned int i)`

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5; 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.
OF B2 /r	LSS r16,m16:16	RM	Valid	Valid	Load SS:r16 with far pointer from memory.
OF B2 /r	LSS r32,m16:32	RM	Valid	Valid	Load SS:r32 with far pointer from memory.
REX + OF 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.
OF B4 /r	LFS r16,m16:16	RM	Valid	Valid	Load FS:r16 with far pointer from memory.
OF B4 /r	LFS r32,m16:32	RM	Valid	Valid	Load FS:r32 with far pointer from memory.
REX + OF B4 /r	LFS r64,m16:64	RM	Valid	N.E.	Load FS:r64 with far pointer from memory.
OF B5 /r	LGS r16,m16:16	RM	Valid	Valid	Load GS:r16 with far pointer from memory.
OF B5 /r	LGS r32,m16:32	RM	Valid	Valid	Load GS:r32 with far pointer from memory.
REX + OF 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)	NA	NA

### 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(0); Fl;
      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); Fl;
      ELSE IF Segment marked not present
        THEN #SS(selector); Fl;
      Fl;
      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); Fl;
      ELSE IF Segment marked not present
        THEN #NP(selector); Fl;
      Fl;
      SegmentRegister ← SegmentSelector(SRC) ;
      SegmentRegister ← SegmentDescriptor([SRC]);
    Fl;
  ELSE IF FS, or GS is loaded with a NULL selector:
    THEN
      SegmentRegister ← NULLSelector;
      SegmentRegister(DescriptorValidBit) ← 0; Fl; (* Hidden flag;
          not accessible by software *)
    Fl;
  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); Fl;
      ELSE IF Segment marked not present
        THEN #SS(selector); Fl;
      Fl;

```

```

SS ← SegmentSelector(SRC);
SS ← SegmentDescriptor([SRC]);
ELSE IF DS, ES, FS, or GS is loaded with non-NUL 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); Fl;
    ELSE IF Segment marked not present
        THEN #NP(selector); Fl;
    Fl;
    SegmentRegister ← SegmentSelector(SRC) AND RPL;
    SegmentRegister ← SegmentDescriptor([SRC]);
Fl;
ELSE IF DS, ES, FS, or GS is loaded with a NUL selector:
    THEN
        SegmentRegister ← NULLSelector;
        SegmentRegister(DescriptorValidBit) ← 0; Fl; (* Hidden flag;
            not accessible by software *)
    Fl;
    DEST ← Offset(SRC);

```

#### Real-Address or Virtual-8086 Mode

```

SegmentRegister ← SegmentSelector(SRC); Fl;
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 NUL 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 NUL 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-NUL 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 nonconforming-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-NUL 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.

## 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 <i>m</i> in register <i>r16</i> .
8D /r	LEA r32,m	RM	Valid	Valid	Store effective address for <i>m</i> in register <i>r32</i> .
REX.W + 8D /r	LEA r64,m	RM	Valid	N.E.	Store effective address for <i>m</i> in register <i>r64</i> .

### Instruction Operand Encoding

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

### 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 processor's 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 3-63. 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 3-64.

**Table 3-64. 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	NP	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	NP	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	NP	Valid	N.E.	Set RSP to RBP, then pop RBP.

### Instruction Operand Encoding

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

### 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 7 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-Bit Mode	Compat/ Leg Mode	Description
OF AE /5	LFENCE	NP	Valid	Valid	Serializes load operations.

**Instruction Operand Encoding**

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

**Description**

Performs a serializing operation on all 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 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.

**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[bit 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 <i>m</i> 16&32	M	N.E.	Valid	Load <i>m</i> into GDTR.
0F 01 /3	LIDT <i>m</i> 16&32	M	N.E.	Valid	Load <i>m</i> into IDTR.
0F 01 /2	LGDT <i>m</i> 16&64	M	Valid	N.E.	Load <i>m</i> into GDTR.
0F 01 /3	LIDT <i>m</i> 16&64	M	Valid	N.E.	Load <i>m</i> into IDTR.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### 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, *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 00FFFFFF;
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 source operand is not a memory location.  
                  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 source 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**

- #UD                If source operand is not a memory location.  
                  If the LOCK prefix is used.
- #GP(0)            The LGDT and LIDT instructions are not recognized in virtual-8086 mode.
- #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 source operand is not a memory location.
- If the LOCK prefix is used.
- #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
OF 00 /2	LLDT r/m16	M	Valid	Valid	Load segment selector <i>r/m16</i> into LDTR.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

**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 points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. 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 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause 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 should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

**Operation**

IF SRC(Offset) > descriptor table limit  
     THEN #GP(segment selector); Fl;

IF segment selector is valid

    Read segment descriptor;

    IF SegmentDescriptor(Type) ≠ LDT  
         THEN #GP(segment selector); Fl;

    IF segment descriptor is not present  
         THEN #NP(segment selector); Fl;

    LDTR(SegmentSelector) ← SRC;  
     LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;  
  ELSE LDTR ← INVALID  
  Fl;

**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 contains a NULL segment selector.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
	Segment selector is beyond 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 selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
	Segment selector is beyond 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
OF 01 /6	LMSW r/m16	M	Valid	Valid	Loads r/m16 in machine status word of CRO.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

**Description**

Loads the source operand into the machine status word, bits 0 through 15 of register CRO. 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 CRO. The PG, CD, NW, AM, WP, NE, and ET flags of CRO 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 the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the whole CRO register. The MOV CRO instruction can be used to set and clear the PE flag in CRO, 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 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

CRO[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)            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 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.

**LOCK—Assert LOCK# Signal Prefix**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F0	LOCK	NP	Valid	Valid	Asserts LOCK# signal for duration of the accompanying instruction.

**NOTES:**

\* See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

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

**Description**

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.

Note that, in later Intel 64 and IA-32 processors (including the Pentium 4, Intel Xeon, and P6 family processors), locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below.

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 8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the for more information on locking of caches.

**Operation**

AssertLOCK#(DurationOfAccompanyingInstruction);

**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**

<b>Opcode</b>	<b>Instruction</b>	<b>Op/ En</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
AC	LODS <i>m8</i>	NP	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 <i>m16</i>	NP	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 <i>m32</i>	NP	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 <i>m64</i>	NP	Valid	N.E.	Load qword at address (R)SI into RAX.
AC	LODSB	NP	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	NP	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	NP	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	NP	Valid	N.E.	Load qword at address (R)SI into RAX.

**Instruction Operand Encoding**

<b>Op/En</b>	<b>Operand 1</b>	<b>Operand 2</b>	<b>Operand 3</b>	<b>Operand 4</b>
NP	NA	NA	NA	NA

**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/REPE/REPZ /REPNE/REPNZ—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;
    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;

```

## 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/LOOPcc—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	NA	NA	NA

### 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 (LOOPcc) 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 LOOPcc 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;
```

```

        Fl;
        Fl;
ELSE (* Instruction = LOOP *)
    IF (Count ≠ 0)
        THEN BranchCond ← 1;
        ELSE BranchCond ← 0;
    Fl;
Fl;

IF BranchCond = 1
THEN
    IF OperandSize = 32
        THEN EIP ← EIP + SignExtend(DEST);
    ELSE IF OperandSize = 64
        THEN RIP ← RIP + SignExtend(DEST);
    Fl;
    ELSE IF OperandSize = 16
        THEN EIP ← EIP AND 0000FFFFH;
    Fl;
Fl;
IF OperandSize = (32 or 64)
    THEN IF (R/E)IP < CS.Base or (R/E)IP > CS.Limit
        #GP; Fl;
    Fl;
Fl;
ELSE
    Terminate loop and continue program execution at (R/E)IP;
Fl;

```

### 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
OF 03 /r	LSL $r16, r16/m16$	RM	Valid	Valid	Load: $r16 \leftarrow$ segment limit, selector $r16/m16$ .
OF 03 /r	LSL $r32, r32/m16^*$	RM	Valid	Valid	Load: $r32 \leftarrow$ segment limit, selector $r32/m16$ .
REX.W + OF 03 /r	LSL $r64, r32/m16^*$	RM	Valid	Valid	Load: $r64 \leftarrow$ segment limit, selector $r32/m16$

### NOTES:

\* For all loads (regardless of destination sizing), only bits 16-0 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)	NA	NA

### Description

Loads the unscrambled segment limit from the segment descriptor 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. 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). 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 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

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 following table.
- 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 3-65. Segment and Gate Descriptor Types**

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Upper 8 byte of a 16-Byte descriptor	Yes
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	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	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	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

## Operation

IF SRC(Offset) > descriptor table limit

THEN ZF ← 0; Fl;

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 (G ← 1)

THEN temp ← ShiftLeft(12, temp) OR 00000FFFH;

ELSE IF OperandSize = 32

THEN DEST ← temp; Fl;

ELSE IF OperandSize = 64 (\* REX.W used \*)

THEN DEST (\* Zero-extended \*) ← temp; Fl;

ELSE (\* OperandSize = 16 \*)

DEST ← temp AND FFFFH;

Fl;

Fl;

## Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set 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 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
OF 00 /3	LTR <i>r/m16</i>	M	Valid	Valid	Load <i>r/m16</i> into task register.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

**Description**

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

**Operation**

IF SRC is a NULL selector  
THEN #GP(0);

IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global  
THEN #GP(segment selector); Fl;

Read segment descriptor;

IF segment descriptor is not for an available TSS  
THEN #GP(segment selector); Fl;

IF segment descriptor is not present  
THEN #NP(segment selector); Fl;

TSSsegmentDescriptor(busy) ← 1;  
(\* Locked read-modify-write operation on the entire descriptor when setting busy flag \*)

TaskRegister(SegmentSelector) ← SRC;  
TaskRegister(SegmentDescriptor) ← TSSSegmentDescriptor;

**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 it contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy. If the selector points to LDT or is beyond the GDT limit.
#NP(selector)	If the TSS 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.

## 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 or to one for a task that is already busy. If the selector points to 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 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**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 -bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
F3 OF 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 OF BD /r LZCNT r32, r/m32	RM	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
REX.W + F3 OF 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**

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

**Description**

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT differs from BSR. For example, LZCNT will produce the operand size when the input operand is zero. It should be noted that on processors that do not support LZCNT, the instruction byte encoding is executed as BSR. In 64-bit mode 64-bit operand size requires REX.W=1.

**Operation**

```

temp ← OperandSize - 1
DEST ← 0
WHILE (temp >= 0) AND (Bit(SRC, temp) = 0)
DO
    temp ← temp - 1
    DEST ← DEST+ 1
OD
IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI
IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI

```

**Flags Affected**

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

## Intel C/C++ Compiler Intrinsic Equivalent

LZCNT:      `unsigned __int32 _lzcnt_u32(unsigned __int32 src);`  
 LZCNT:      `unsigned __int64 _lzcnt_u64(unsigned __int64 src);`

## Protected Mode Exceptions

- #GP(0)            For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  
                   If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
- #SS(0)            For an illegal address in the SS segment.
- #PF (fault-code)    For a page fault.
- #AC(0)            If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

- #GP(0)            If any part of the operand lies outside of the effective address space from 0 to OFFFFH.
- #SS(0)            For an illegal address in the SS segment.

## Virtual 8086 Mode Exceptions

- #GP(0)            If any part of the operand lies outside of the effective address space from 0 to OFFFFH.
- #SS(0)            For an illegal address in the SS segment.
- #PF (fault-code)    For a page fault.
- #AC(0)            If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## 64-Bit Mode Exceptions

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

**MASKMOVDQU—Store Selected Bytes of Double Quadword**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F F7 /r MASKMOVDQU <i>xmm1, xmm2</i>		RM V/V	SSE2	Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> . The default memory location is specified by DS:DI/EDI/RDI.
VEX.128.66.0F.WIG F7 /r VMASKMOVDQU <i>xmm1, xmm2</i>		RM V/V	AVX	Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> . The default memory location is specified by DS:DI/EDI/RDI.

**Instruction Operand Encoding<sup>1</sup>**

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

**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 *); Fl;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); Fl;
    (* 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 *); Fl;

```

## Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)
```

## Other Exceptions

See Exceptions Type 4; 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
OF F7 /r MASKMOVQ mm1, mm2	RM	Valid	Valid	Selectively write bytes from <i>mm1</i> to memory location using the byte mask in <i>mm2</i> . 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)	NA	NA

**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 7*). 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 Os [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 *); Fl;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); Fl;
    (* 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 *); Fl;

```

## Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmove_si64(__m64d, __m64n, char * p)
```

## Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**MAXPD—Return Maximum Packed Double-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5F /r MAXPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Return the maximum double-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 5F /r VMAXPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the maximum double-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5F /r VMAXPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the maximum packed double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

**Description**

Performs an SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

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

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

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

**Operation****MAX(SRC1, SRC2)**

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}
```

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

$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{MAX}(\text{DEST}[127:64], \text{SRC}[127:64])$   
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

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

$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64])$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

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

$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64])$   
 $\text{DEST}[191:128] \leftarrow \text{MAX}(\text{SRC1}[191:128], \text{SRC2}[191:128])$   
 $\text{DEST}[255:192] \leftarrow \text{MAX}(\text{SRC1}[255:192], \text{SRC2}[255:192])$

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

MAXPD:      `__m128d _mm_max_pd(__m128d a, __m128d b);`  
 VMAXPD:     `__m256d _mm256_max_pd (__m256d a, __m256d b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

**Other Exceptions**

See Exceptions Type 2.

**MAXPS—Return Maximum Packed Single-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5F /r MAXPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Return the maximum single-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 5F /r VMAXPS <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the maximum single-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 5F /r VMAXPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the maximum single double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

**Instruction Operand Encoding**

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

**Description**

Performs an SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

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

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

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

**Operation****MAX(SRC1, SRC2)**

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
}
FI;
```

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

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{DEST}[31:0], \text{SRC}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{DEST}[63:32], \text{SRC}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{DEST}[95:64], \text{SRC}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{DEST}[127:96], \text{SRC}[127:96])$   
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

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

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{SRC1}[63:32], \text{SRC2}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{SRC1}[95:64], \text{SRC2}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{SRC1}[127:96], \text{SRC2}[127:96])$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

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

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{SRC1}[63:32], \text{SRC2}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{SRC1}[95:64], \text{SRC2}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{SRC1}[127:96], \text{SRC2}[127:96])$   
 $\text{DEST}[159:128] \leftarrow \text{MAX}(\text{SRC1}[159:128], \text{SRC2}[159:128])$   
 $\text{DEST}[191:160] \leftarrow \text{MAX}(\text{SRC1}[191:160], \text{SRC2}[191:160])$   
 $\text{DEST}[223:192] \leftarrow \text{MAX}(\text{SRC1}[223:192], \text{SRC2}[223:192])$   
 $\text{DEST}[255:224] \leftarrow \text{MAX}(\text{SRC1}[255:224], \text{SRC2}[255:224])$

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

MAXPS:       `__m128 _mm_max_ps (__m128 a, __m128 b);`  
 VMAXPS:      `__m256 _mm256_max_ps (__m256 a, __m256 b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

**Other Exceptions**

See Exceptions Type 2.

**MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Return the maximum scalar double-precision floating-point value between <i>xmm2/mem64</i> and <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5F /r VMAXSD <i>xmm1, xmm2, xmm3/m64</i>	RVM	V/V	AVX	Return the maximum scalar double-precision floating-point value between <i>xmm3/mem64</i> and <i>xmm2</i> .

**Instruction Operand Encoding**

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

**Description**

Compares the low double-precision floating-point values in the first source operand and second the source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed. The high quadword of the destination operand is copied from the same bits of first source operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

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

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

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation**

**MAX(SRC1, SRC2)**

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; F1;
  ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; F1;
  ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  F1;
}
```

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

DEST[63:0]  $\leftarrow \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0])$   
DEST[VLMAX-1:64] (Unmodified)

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

DEST[63:0]  $\leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$   
DEST[127:64]  $\leftarrow \text{SRC1}[127:64]$   
DEST[VLMAX-1:128]  $\leftarrow 0$

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

MAXSD:       `__m128d _mm_max_sd(__m128d a, __m128d b)`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

**Other Exceptions**

See Exceptions Type 3.

**MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5F /r MAXSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Return the maximum scalar single-precision floating-point value between <i>xmm2/mem32</i> and <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 5F /r VMAXSS <i>xmm1, xmm2, xmm3/m32</i>	RVM	V/V	AVX	Return the maximum scalar single-precision floating-point value between <i>xmm3/mem32</i> and <i>xmm2</i> .

**Instruction Operand Encoding**

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

**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.

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

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

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation****MAX(SRC1, SRC2)**

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}
```

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

DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])  
DEST[VLMAX-1:32] (Unmodified)

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

DEST[31:0]  $\leftarrow \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0])$   
DEST[127:32]  $\leftarrow \text{SRC1}[127:32]$   
DEST[VLMAX-1:128]  $\leftarrow 0$

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

`__m128d _mm_max_ss(__m128d a, __m128d b)`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

**Other Exceptions**

See Exceptions Type 3.

**MFENCE—Memory Fence**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /6	MFENCE	NP	Valid	Valid	Serializes load and store operations.

**Instruction Operand Encoding**

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

**Description**

Performs a serializing operation on all 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.<sup>1</sup> 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.

**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[bit 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—Return Minimum Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5D /r MINPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Return the minimum double-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 5D /r VMINPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the minimum double-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5D /r VMINPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the minimum packed double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

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

### Description

Performs an SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

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

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

### Operation

```

MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; F1;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; F1;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    F1;
}

```

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

$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$   
 $\text{DEST}[\text{VLMAX}-1:128] \text{ (Unmodified)}$

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

$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

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

$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$   
 $\text{DEST}[191:128] \leftarrow \text{MIN}(\text{SRC1}[191:128], \text{SRC2}[191:128])$   
 $\text{DEST}[255:192] \leftarrow \text{MIN}(\text{SRC1}[255:192], \text{SRC2}[255:192])$

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

MINPD:      `__m128d _mm_min_pd(__m128d a, __m128d b);`  
 VMINPD:     `__m256d _mm256_min_pd (__m256d a, __m256d b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

**Other Exceptions**

See Exceptions Type 2.

## MINPS—Return Minimum Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5D /r MINPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Return the minimum single-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 5D /r VMINPS <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the minimum single-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 5D /r VMINPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the minimum single double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

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

### Description

Performs an SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

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

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

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

### Operation

```

MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
}

```

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

$\text{DEST}[31:0] \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MIN}(\text{SRC1}[63:32], \text{SRC2}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MIN}(\text{SRC1}[95:64], \text{SRC2}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MIN}(\text{SRC1}[127:96], \text{SRC2}[127:96])$   
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

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

$\text{DEST}[31:0] \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MIN}(\text{SRC1}[63:32], \text{SRC2}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MIN}(\text{SRC1}[95:64], \text{SRC2}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MIN}(\text{SRC1}[127:96], \text{SRC2}[127:96])$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

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

$\text{DEST}[31:0] \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MIN}(\text{SRC1}[63:32], \text{SRC2}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MIN}(\text{SRC1}[95:64], \text{SRC2}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MIN}(\text{SRC1}[127:96], \text{SRC2}[127:96])$   
 $\text{DEST}[159:128] \leftarrow \text{MIN}(\text{SRC1}[159:128], \text{SRC2}[159:128])$   
 $\text{DEST}[191:160] \leftarrow \text{MIN}(\text{SRC1}[191:160], \text{SRC2}[191:160])$   
 $\text{DEST}[223:192] \leftarrow \text{MIN}(\text{SRC1}[223:192], \text{SRC2}[223:192])$   
 $\text{DEST}[255:224] \leftarrow \text{MIN}(\text{SRC1}[255:224], \text{SRC2}[255:224])$

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

MINPS:       `__m128d _mm_min_ps(__m128d a, __m128d b);`  
 VMINPS:      `__m256 _mm256_min_ps (__m256 a, __m256 b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

**Other Exceptions**

See Exceptions Type 2.

**MINSD—Return Minimum Scalar Double-Precision Floating-Point Value**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 5D /r MINSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Return the minimum scalar double-precision floating-point value between <i>xmm2/mem64</i> and <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5D /r VMINSD <i>xmm1, xmm2, xmm3/m64</i>	RVM	V/V	AVX	Return the minimum scalar double precision floating-point value between <i>xmm3/mem64</i> and <i>xmm2</i> .

**Instruction Operand Encoding**

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

**Description**

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed. The high quadword of the destination operand is copied from the same bits in the first source operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of 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.

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

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

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation**

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; Fl;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; Fl;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    Fl;
}
```

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

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])  
DEST[VLMAX-1:64] (Unmodified)

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

DEST[63:0]  $\leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$   
DEST[127:64]  $\leftarrow \text{SRC1}[127:64]$   
DEST[VLMAX-1:128]  $\leftarrow 0$

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

MINSD:        $\_\_m128d \_\_mm\_min\_sd(\_\_m128d a, \_\_m128d b)$

**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

**Other Exceptions**

See Exceptions Type 3.

## MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5D /r MINSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Return the minimum scalar single-precision floating-point value between <i>xmm2/mem32</i> and <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.WIG 5D /r VMINSS <i>xmm1,xmm2, xmm3/m32</i>	RVM	V/V	AVX	Return the minimum scalar single precision floating-point value between <i>xmm3/mem32</i> and <i>xmm2</i> .

### Instruction Operand Encoding

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

### 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.

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

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

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

```

MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

```

### MINSS (128-bit Legacy SSE version)

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])  
DEST[VLMAX-1:32] (Unmodified)

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

DEST[31:0]  $\leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$   
DEST[127:32]  $\leftarrow \text{SRC1}[127:32]$   
DEST[VLMAX-1:128]  $\leftarrow 0$

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

MINSS:       `__m128d _mm_min_ss(__m128d a, __m128d b)`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

**Other Exceptions**

See Exceptions Type 3.

## MONITOR—Set Up Monitor Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 C8	MONITOR	NP	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:EAX (DS:RAX in 64-bit mode).

### Instruction Operand Encoding

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

### 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 content of EAX is an effective address (in 64-bit mode, RAX is used). 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 8, “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[bit 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[bit 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[bit 3] = 0.

### Virtual 8086 Mode Exceptions

#UD	The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H: ECX.MONITOR[bit 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[bit 3] = 0.

## MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8,r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8***,r8***	MR	Valid	N.E.	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,r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8***,r/m8***	RM	Valid	N.E.	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**	MR	Valid	Valid	Move segment register to r/m16.
REX.W + 8C /r	MOV r/m64,Sreg**	MR	Valid	Valid	Move zero extended 16-bit segment register to r/m64.
8E /r	MOV Sreg,r/m16**	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg,r/m64**	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL,moffs8*	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL,moffs8*	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX,moffs16*	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32*	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX,moffs64*	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***,AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16*,AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32*,EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64*,RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8,imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8***,imm8	OI	Valid	N.E.	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,imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /0 ib	MOV r/m8***,imm8	MI	Valid	N.E.	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 io	MOV r/m64,imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

**NOTES:**

- \* 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.
- \*\* 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).
- \*\*\*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

**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 inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an interrupt occurs<sup>1</sup>. Be aware that the LSS instruction offers a more efficient method of loading the SS and ESP registers.

When operating in 32-bit mode and moving data between a segment register and a general-purpose register, the 32-bit IA-32 processors do not require the use of the 16-bit operand-size prefix (a byte with the value 66H) with

---

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

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

```
MOV SS, EDX
MOV SS, EAX
MOV ESP, EBP
```

this instruction, but most assemblers will insert it if the standard form of the instruction is used (for example, MOV DS, AX). The processor will execute this instruction correctly, but it will usually require an extra clock. With most assemblers, using the instruction form MOV DS, EAX will avoid this unneeded 66H prefix. When the processor executes the instruction with a 32-bit general-purpose register, it assumes that the 16 least-significant bits of the general-purpose register are the destination or source operand. If the register is a destination operand, the resulting value in the two high-order bytes of the register is implementation dependent. For the Pentium 4, Intel Xeon, and P6 family processors, the two high-order bytes are filled with zeros; for earlier 32-bit IA-32 processors, the two high order bytes are undefined.

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); Fl;

  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); Fl;

  IF segment not marked present

    THEN #SS(selector);

  ELSE

    SS ← segment selector;

    SS ← segment descriptor; Fl;

Fl;

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)

    or ((RPL > DPL) and (CPL > DPL))

      THEN #GP(selector); Fl;

  IF segment not marked present

    THEN #NP(selector);

  ELSE

    SegmentRegister ← segment selector;

    SegmentRegister ← segment descriptor; Fl;

Fl;

IF DS, ES, FS, or GS is loaded with NULL selector

THEN

  SegmentRegister ← segment selector;

  SegmentRegister ← segment descriptor;

Fl;

**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, but both the RPL and the CPL are 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.
  - If the stack address is in a non-canonical form.
- #SS(0)
- #SS(selector)
- #PF(fault-code)
- #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
OF 20/r	MR	N.E.	Valid	Move control register to <i>r</i> 32.
MOV <i>r</i> 32, CR0-CR7				
OF 20/r	MR	Valid	N.E.	Move extended control register to <i>r</i> 64.
MOV <i>r</i> 64, CR0-CR7				
REX.R + OF 20 /0	MR	Valid	N.E.	Move extended CR8 to <i>r</i> 64. <sup>1</sup>
MOV <i>r</i> 64, CR8				
OF 22 /r	RM	N.E.	Valid	Move <i>r</i> 32 to control register.
MOV CR0-CR7, <i>r</i> 32				
OF 22 /r	RM	Valid	N.E.	Move <i>r</i> 64 to extended control register.
MOV CR0-CR7, <i>r</i> 64				
REX.R + OF 22 /0	RM	Valid	N.E.	Move <i>r</i> 64 to extended CR8. <sup>1</sup>
MOV CR8, <i>r</i> 64				

**NOTE:**

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 8 in *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)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### 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. (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. Attempts to reference CR1, CR5, CR6, CR7, and CR9–CR15 result in undefined opcode (#UD) exceptions.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4’s reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. 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.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, “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 4.10.4.1, “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 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

$\text{DEST} \leftarrow \text{SRC};$

## 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.
#UD	If the LOCK prefix is used. If an attempt is made to access CR1, CR5, CR6, or CR7.

## 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).
#UD	If the LOCK prefix is used. If an attempt is made to access CR1, CR5, CR6, or CR7.

## 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 write a 1 to any reserved bit in CR3. 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, or CR7.

**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.
- 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, or CR7.
- 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
OF 21/r	MR	N.E.	Valid	Move debug register to <i>r</i> 32.
MOV <i>r</i> 32, DR0-DR7				
OF 21/r	MR	Valid	N.E.	Move extended debug register to <i>r</i> 64.
MOV <i>r</i> 64, DR0-DR7				
OF 23 /r	RM	N.E.	Valid	Move <i>r</i> 32 to debug register.
MOV DR0-DR7, <i>r</i> 32				
OF 23 /r	RM	Valid	N.E.	Move <i>r</i> 64 to extended debug register.
MOV DR0-DR7, <i>r</i> 64				

### Instruction Operand Encoding

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

### 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 17.2, “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;
```

Fl;

### 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	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 29 /r MOVAPD <i>xmm2/m128, xmm1</i>	MR	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 28 /r VMOVAPD <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move aligned packed double-precision floating-point values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 29 /r VMOVAPD <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move aligned packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 28 /r VMOVAPD <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move aligned packed double-precision floating-point values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.66.0F.WIG 29 /r VMOVAPD <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move aligned packed double-precision floating-point values from <i>ymm1</i> to <i>ymm2/mem</i> .

### Instruction Operand Encoding

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

### Description

Moves 2 or 4 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated.

To move double-precision floating-point values to and from unaligned memory locations, use the (V)MOVUPD instruction.

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

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

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

VEX.256 encoded version: Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

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

## Operation

### **MOVAPD (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128] (Unmodified)

### **(V)MOVAPD (128-bit store-form version)**

DEST[127:0]  $\leftarrow$  SRC[127:0]

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

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VMOVAPD (VEX.256 encoded version)**

DEST[255:0]  $\leftarrow$  SRC[255:0]

## Intel C/C++ Compiler Intrinsic Equivalent

```
MOVAPD:    __m128d _mm_load_pd (double const * p);
MOVAPD:    _mm_store_pd(double * p, __m128d a);
VMOVAPD:   __m256d _mm256_load_pd (double const * p);
VMOVAPD:   _mm256_store_pd(double * p, __m256d a);
```

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 1.SSE2; additionally  
#UD                    If VEX.vvvv != 1111B.

## MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 28 /r MOVAPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Move packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
OF 29 /r MOVAPS <i>xmm2/m128, xmm1</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.0F.WIG 28 /r VMOVAPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move aligned packed single-precision floating-point values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.0F.WIG 29 /r VMOVAPS <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move aligned packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.0F.WIG 28 /r VMOVAPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move aligned packed single-precision floating-point values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.0F.WIG 29 /r VMOVAPS <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move aligned packed single-precision floating-point values from <i>ymm1</i> to <i>ymm2/mem</i> .

### Instruction Operand Encoding

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

### Description

Moves 4 or 8 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated.

To move single-precision floating-point values to and from unaligned memory locations, use the (V)MOVUPS instruction.

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.

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

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

VEX.256 encoded version:

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.

## Operation

### **MOVAPS (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128] (Unmodified)

### **(V)MOVAPS (128-bit store form)**

DEST[127:0]  $\leftarrow$  SRC[127:0]

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

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VMOVAPS (VEX.256 encoded version)**

DEST[255:0]  $\leftarrow$  SRC[255:0]

## Intel C/C++ Compiler Intrinsic Equivalent

```
MOVAPS:      __m128 _mm_load_ps (float const * p);
MOVAPS:      _mm_store_ps(float * p, __m128 a);
VMOVAPS:     __m256 _mm256_load_ps (float const * p);
VMOVAPS:     _mm256_store_ps(float * p, __m256 a);
```

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 1.SSE; additionally

#UD                    If VEX.vvvv != 1111B.

## MOVBE—Move Data After Swapping Bytes

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 38 F0 /r	MOVBE <i>r16, m16</i>	RM	Valid	Valid	Reverse byte order in <i>m16</i> and move to <i>r16</i> .
OF 38 F0 /r	MOVBE <i>r32, m32</i>	RM	Valid	Valid	Reverse byte order in <i>m32</i> and move to <i>r32</i> .
REX.W + OF 38 F0 /r	MOVBE <i>r64, m64</i>	RM	Valid	N.E.	Reverse byte order in <i>m64</i> and move to <i>r64</i> .
OF 38 F1 /r	MOVBE <i>m16, r16</i>	MR	Valid	Valid	Reverse byte order in <i>r16</i> and move to <i>m16</i> .
OF 38 F1 /r	MOVBE <i>m32, r32</i>	MR	Valid	Valid	Reverse byte order in <i>r32</i> and move to <i>m32</i> .
REX.W + OF 38 F1 /r	MOVBE <i>m64, r64</i>	MR	Valid	N.E.	Reverse byte order in <i>r64</i> and move to <i>m64</i> .

### Instruction Operand Encoding

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

### 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[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) 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[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) 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[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used. If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

**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[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

## MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 6E /r		RM	V/V	MMX Move doubleword from <i>r/m32</i> to <i>mm</i> .
MOVD <i>mm</i> , <i>r/m32</i>				
REX.W + OF 6E /r		RM	V/N.E.	MMX Move quadword from <i>r/m64</i> to <i>mm</i> .
MOVQ <i>mm</i> , <i>r/m64</i>				
OF 7E /r		MR	V/V	MMX Move doubleword from <i>mm</i> to <i>r/m32</i> .
MOVD <i>r/m32</i> , <i>mm</i>				
REX.W + OF 7E /r		MR	V/N.E.	MMX Move quadword from <i>mm</i> to <i>r/m64</i> .
MOVQ <i>r/m64</i> , <i>mm</i>				
VEX.128.66.0F.W0 6E /		RM	V/V	AVX Move doubleword from <i>r/m32</i> to <i>xmm1</i> .
VMOVD <i>xmm1</i> , <i>r32/m32</i>				
VEX.128.66.0F.W1 6E /r		RM	V/N.E.	AVX Move quadword from <i>r/m64</i> to <i>xmm1</i> .
VMOVQ <i>xmm1</i> , <i>r64/m64</i>				
66 OF 6E /r		RM	V/V	SSE2 Move doubleword from <i>r/m32</i> to <i>xmm</i> .
MOVD <i>xmm</i> , <i>r/m32</i>				
66 REX.W OF 6E /r		RM	V/N.E.	SSE2 Move quadword from <i>r/m64</i> to <i>xmm</i> .
MOVQ <i>xmm</i> , <i>r/m64</i>				
66 OF 7E /r		MR	V/V	SSE2 Move doubleword from <i>xmm</i> register to <i>r/m32</i> .
MOVD <i>r/m32</i> , <i>xmm</i>				
66 REX.W OF 7E /r		MR	V/N.E.	SSE2 Move quadword from <i>xmm</i> register to <i>r/m64</i> .
MOVQ <i>r/m64</i> , <i>xmm</i>				
VEX.128.66.0F.W0 7E /r		MR	V/V	AVX Move doubleword from <i>xmm1</i> register to <i>r/m32</i> .
VMOVD <i>r32/m32</i> , <i>xmm1</i>				
VEX.128.66.0F.W1 7E /r		MR	V/N.E.	AVX Move quadword from <i>xmm1</i> register to <i>r/m64</i> .
VMOVQ <i>r64/m64</i> , <i>xmm1</i>				

### Instruction Operand Encoding

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

### 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.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

### **MOVD (when destination operand is MMX technology register)**

DEST[31:0] ← SRC;  
DEST[63:32] ← 00000000H;

### **MOVD (when destination operand is XMM register)**

DEST[31:0] ← SRC;  
DEST[127:32] ← 000000000000000000000000000000H;  
DEST[VLMAX-1:128] (Unmodified)

### **MOVD (when source operand is 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[VLMAX-1:32] ← 0

### **MOVQ (when destination operand is XMM register)**

DEST[63:0] ← SRC[63:0];  
DEST[127:64] ← 0000000000000000H;  
DEST[VLMAX-1:128] (Unmodified)

### **MOVQ (when destination operand is r/m64)**

DEST[63:0] ← SRC[63:0];

### **MOVQ (when source operand is 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[VLMAX-1:64] ← 0

## Intel C/C++ Compiler Intrinsic Equivalent

MOVD:	<code>__m64 _mm_cvtsi32_si64 (int i )</code>
MOVD:	<code>int _mm_cvtsi64_si32 ( __m64m )</code>
MOVD:	<code>__m128i _mm_cvtsi32_si128 (int a)</code>
MOVD:	<code>int _mm_cvtsi128_si32 ( __m128i a)</code>
MOVQ:	<code>__int64 _mm_cvtsi128_si64(__m128i);</code>
MOVQ:	<code>__m128i _mm_cvtsi64_si128(__int64);</code>

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 5; additionally

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

**MOVDDUP—Move One Double-FP and Duplicate**

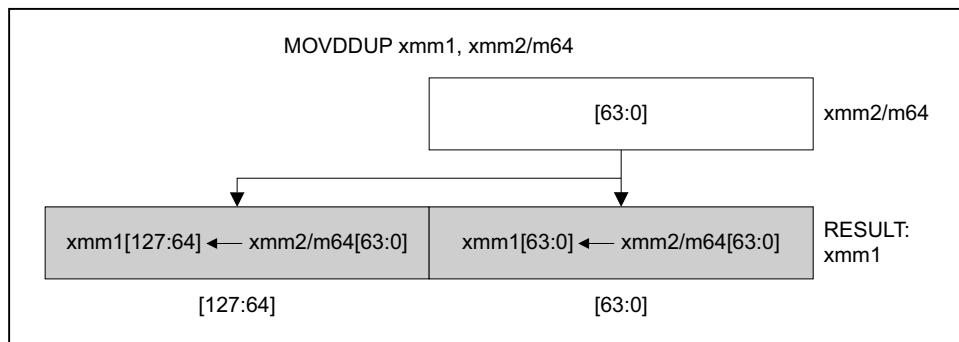
Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 12 /r MOVDDUP <i>xmm1, xmm2/m64</i>	RM	V/V	SSE3	Move one double-precision floating-point value from the lower 64-bit operand in <i>xmm2/m64</i> to <i>xmm1</i> and duplicate.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Move double-precision floating-point values from <i>xmm2/mem</i> and duplicate into <i>xmm1</i> .
VEX.256.F2.0F.WIG 12 /r VMOVDDUP <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move even index double-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

**Instruction Operand Encoding**

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

**Description**

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 8 bytes of data at memory location m64 are loaded. When the register-register form of this operation is used, the lower half of the 128-bit source register is duplicated and copied into the 128-bit destination register. See Figure 3-24.



OM15997

**Figure 3-24. MOVDDUP—Move One Double-FP and Duplicate**

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

## Operation

```

IF (Source = m64)
    THEN
        (* Load instruction *)
        xmm1[63:0] = m64;
        xmm1[127:64] = m64;
    ELSE
        (* Move instruction *)
        xmm1[63:0] = xmm2[63:0];
        xmm1[127:64] = xmm2[63:0];
FI;

```

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

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[\text{VLMAX}-1:128] \text{ (Unmodified)}$

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

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

### **VMOVDDUP (VEX.256 encoded version)**

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[191:128] \leftarrow \text{SRC}[191:128]$   
 $\text{DEST}[255:192] \leftarrow \text{SRC}[191:128]$

## Intel C/C++ Compiler Intrinsic Equivalent

MOVDDUP:	<code>__m128d _mm_movedup_pd(__m128d a)</code>
MOVDDUP:	<code>__m128d _mm_loaddup_pd(double const * dp)</code>

## SIMD Floating-Point Exceptions

None

## Other Exceptions

See Exceptions Type 5; additionally

#UD                    If VEX.vvvv != 1111B.

**MOVDQA—Move Aligned Double Quadword**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32-bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
66 0F 6F /r MOVDQA <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Move aligned double quadword from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 7F /r MOVDQA <i>xmm2/m128, xmm1</i>	MR	V/V	SSE2	Move aligned double quadword from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 6F /r VMOVDQA <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move aligned packed integer values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 7F /r VMOVDQA <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move aligned packed integer values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 6F /r VMOVDQA <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move aligned packed integer values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.66.0F.WIG 7F /r VMOVDQA <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move aligned packed integer values from <i>ymm1</i> to <i>ymm2/mem</i> .

**Instruction Operand Encoding**

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

**Description**

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.

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

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

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

VEX.256 encoded version:

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.

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

## Operation

### **MOVQDA (128-bit load- and register- form Legacy SSE version)**

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128] (Unmodified)  
 (\* #GP if SRC or DEST unaligned memory operand \*)

### **(V)MOVQDA (128-bit store forms)**

DEST[127:0]  $\leftarrow$  SRC[127:0]

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

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VMOVQDA (VEX.256 encoded version)**

DEST[255:0]  $\leftarrow$  SRC[255:0]

## Intel C/C++ Compiler Intrinsic Equivalent

MOVQDA:	<code>__m128i _mm_load_si128 ( __m128i *p)</code>
MOVQDA:	<code>void _mm_store_si128 ( __m128i *p, __m128i a)</code>
VMOVQDA:	<code>__m256i _mm256_load_si256 ( __m256i * p);</code>
VMOVQDA:	<code>_mm256_store_si256( __m256i *p, __m256i a);</code>

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD                    If VEX.vvvv != 1111B.

**MOVDQU—Move Unaligned Double Quadword**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32-bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
F3 OF 6F /r MOVDQU <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Move unaligned double quadword from <i>xmm2/m128</i> to <i>xmm1</i> .
F3 OF 7F /r MOVDQU <i>xmm2/m128, xmm1</i>	MR	V/V	SSE2	Move unaligned double quadword from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.F3.0F.WIG 6F /r VMOVDQU <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move unaligned packed integer values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.F3.0F.WIG 7F /r VMOVDQU <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move unaligned packed integer values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.F3.0F.WIG 6F /r VMOVDQU <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move unaligned packed integer values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.F3.0F.WIG 7F /r VMOVDQU <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move unaligned packed integer values from <i>ymm1</i> to <i>ymm2/mem</i> .

**Instruction Operand Encoding**

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

**Description**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 may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.<sup>1</sup>

To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the MOVDQA instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

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

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

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

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

1. 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 operand is not aligned on an 8-byte boundary.

location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

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

## Operation

### **MOVDQU load and register copy (128-bit Legacy SSE version)**

DEST[127:0]  $\leftarrow$  SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

### **(V)MOVDQU 128-bit store-form versions**

DEST[127:0]  $\leftarrow$  SRC[127:0]

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

DEST[127:0]  $\leftarrow$  SRC[127:0]

DEST[VLMAX-1:128]  $\leftarrow$  0

### **VMOVDQU (VEX.256 encoded version)**

DEST[255:0]  $\leftarrow$  SRC[255:0]

## Intel C/C++ Compiler Intrinsic Equivalent

MOVDQU:	void _mm_storeu_si128 ( __m128i *p, __m128i a)
MOVDQU:	__m128i _mm_loadu_si128 ( __m128i *p)
VMOVDQU:	__m256i _mm256_loadu_si256 ( __m256i * p);
VMOVDQU:	_mm256_storeu_si256( __m256i *p, __m256i a);

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD                            If VEX.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 <i>mm, xmm</i>	RM	Valid	Valid	Move low quadword from <i>xmm</i> to <i>mmx</i> register.

**Instruction Operand Encoding**

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

**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 CRO.TS[bit 3] = 1.
- #UD              If CRO.EM[bit 2] = 1.
- If CR4.OSFXSR[bit 9] = 0.
- If CPUID.01H:EDX.SSE2[bit 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.

## MOVHLPS— Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 12 /r MOVHLPS <i>xmm1, xmm2</i>		RM V/V	SSE	Move two packed single-precision floating-point values from high quadword of <i>xmm2</i> to low quadword of <i>xmm1</i> .
VEX.NDS.128.0F.WIG 12 /r VMOVHLPS <i>xmm1, xmm2, xmm3</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of <i>xmm3</i> and low quadword of <i>xmm2</i> .

### Instruction Operand Encoding

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

### Description

This instruction cannot be used for memory to register moves.

#### 128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The high quadword of the destination operand is left unchanged. Bits (VLMAX-1:64) of the corresponding YMM destination register are unmodified.

#### 128-bit three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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

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

### Operation

#### MOVHLPS (128-bit two-argument form)

DEST[63:0]  $\leftarrow$  SRC[127:64]

DEST[VLMAX-1:64] (Unmodified)

#### VMOVHLPS (128-bit three-argument form)

DEST[63:0]  $\leftarrow$  SRC2[127:64]

DEST[127:64]  $\leftarrow$  SRC1[127:64]

DEST[VLMAX-1:128]  $\leftarrow$  0

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

MOVHLPS:  $\text{__m128\_mm\_movehl\_ps}(\text{__m128 a}, \text{__m128 b})$

### SIMD Floating-Point Exceptions

None.

**Other Exceptions**

See Exceptions Type 7; additionally

#UD                    If VEX.L= 1.

## MOVHPD—Move High Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD <i>xmm, m64</i>	RM	V/V	SSE2	Move double-precision floating-point value from <i>m64</i> to high quadword of <i>xmm</i> .
66 0F 17 /r MOVHPD <i>m64, xmm</i>	MR	V/V	SSE2	Move double-precision floating-point value from high quadword of <i>xmm</i> to <i>m64</i> .
VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD <i>xmm2, xmm1, m64</i>	RVM	V/V	AVX	Merge double-precision floating-point value from <i>m64</i> and the low quadword of <i>xmm1</i> .
VEX128.66.0F.WIG 17/r VMOVHPD <i>m64, xmm1</i>	MR	V/V	AVX	Move double-precision floating-point values from high quadword of <i>xmm1</i> to <i>m64</i> .

### Instruction Operand Encoding

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

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

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

#### VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand) are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

#### 128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 OF 17 store. For VMOVHPD (store) (VEX.128.66.0F 17 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

### Operation

#### MOVHPD (128-bit Legacy SSE load)

DEST[63:0] (Unmodified)

DEST[127:64]  $\leftarrow$  SRC[63:0]

DEST[VLMAX-1:128] (Unmodified)

**VMOVHPD (VEX.128 encoded load)**

DEST[63:0]  $\leftarrow$  SRC1[63:0]  
 DEST[127:64]  $\leftarrow$  SRC2[63:0]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

**VMOVHPD (store)**

DEST[63:0]  $\leftarrow$  SRC[127:64]

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

MOVHPD:       `__m128d _mm_loadh_pd ( __m128d a, double *p)`  
 MOVHPD:       `void _mm_storeh_pd (double *p, __m128d a)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5; additionally

#UD              If VEX.L= 1.

## MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 16 /r MOVHPS <i>xmm</i> , <i>m64</i>	RM	V/V	SSE	Move two packed single-precision floating-point values from <i>m64</i> to high quadword of <i>xmm</i> .
OF 17 /r MOVHPS <i>m64</i> , <i>xmm</i>	MR	V/V	SSE	Move two packed single-precision floating-point values from high quadword of <i>xmm</i> to <i>m64</i> .
VEX.NDS.128.0F.WIG 16 /r VMOVHPS <i>xmm2</i> , <i>xmm1</i> , <i>m64</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from <i>m64</i> and the low quadword of <i>xmm1</i> .
VEX.128.0F.WIG 17/r VMOVHPS <i>m64</i> , <i>xmm1</i>	MR	V/V	AVX	Move two packed single-precision floating-point values from high quadword of <i>xmm1</i> to <i>m64</i> .

### Instruction Operand Encoding

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

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

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

#### VEX.128 encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand) are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

#### 128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing OF 17 store. For VMOVHPS (store) (VEX.NDS.128.0F 17 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

## Operation

### **MOVHPS (128-bit Legacy SSE load)**

DEST[63:0] (Unmodified)

DEST[127:64]  $\leftarrow$  SRC[63:0]

DEST[VLMAX-1:128] (Unmodified)

### **VMOVHPS (VEX.128 encoded load)**

DEST[63:0]  $\leftarrow$  SRC1[63:0]

DEST[127:64]  $\leftarrow$  SRC2[63:0]

DEST[VLMAX-1:128]  $\leftarrow$  0

### **VMOVHPS (store)**

DEST[63:0]  $\leftarrow$  SRC[127:64]

## Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS:       `__m128d _mm_loadh_pi ( __m128d a, __m64 *p)`

MOVHPS:       `void _mm_storeh_pi ( __m64 *p, __m128d a)`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 5; additionally

#UD              If VEX.L= 1.

## MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 16 /r MOVLHPS <i>xmm1, xmm2</i>		RM V/V	SSE	Move two packed single-precision floating-point values from low quadword of <i>xmm2</i> to high quadword of <i>xmm1</i> .
VEX.NDS.128.0F.WIG 16 /r VMOVLHPS <i>xmm1, xmm2, xmm3</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of <i>xmm3</i> and low quadword of <i>xmm2</i> .

### Instruction Operand Encoding

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

### Description

This instruction cannot be used for memory to register moves.

#### 128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. The upper 128 bits of the corresponding YMM destination register are unmodified.

#### 128-bit three-argument form

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). The upper 128-bits of the destination YMM register are zeroed.

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

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

### Operation

#### MOVLHPS (128-bit two-argument form)

DEST[63:0] (Unmodified)  
DEST[127:64]  $\leftarrow$  SRC[63:0]  
DEST[VLMAX-1:128] (Unmodified)

#### VMOVLHPS (128-bit three-argument form)

DEST[63:0]  $\leftarrow$  SRC1[63:0]  
DEST[127:64]  $\leftarrow$  SRC2[63:0]  
DEST[VLMAX-1:128]  $\leftarrow$  0

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

MOVHLPS: `__m128 _mm_movehl_ps(__m128 a, __m128 b)`

### SIMD Floating-Point Exceptions

None.

**Other Exceptions**

See Exceptions Type 7; additionally

#UD                    If VEX.L= 1.

## MOVLPD—Move Low Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 OF 12 /r MOVLPD <i>xmm, m64</i>	RM	V/V	SSE2	Move double-precision floating-point value from <i>m64</i> to low quadword of <i>xmm</i> register.
66 OF 13 /r MOVLPD <i>m64, xmm</i>	MR	V/V	SSE2	Move double-precision floating-point nvalue from low quadword of <i>xmm</i> register to <i>m64</i> .
VEX.NDS.128.66.0F.WIG 12 /r VMOVLPD <i>xmm2, xmm1, m64</i>	RVM	V/V	AVX	Merge double-precision floating-point value from <i>m64</i> and the high quadword of <i>xmm1</i> .
VEX.128.66.0F.WIG 13/r VMOVLPD <i>m64, xmm1</i>	MR	V/V	AVX	Move double-precision floating-point values from low quadword of <i>xmm1</i> to <i>m64</i> .

### Instruction Operand Encoding

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

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

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

#### VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). The upper 128-bits of the destination YMM register are zeroed.

#### 128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 OF 13 store. For VMOVLPD (store) (VEX.128.66.0F 13 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

### Operation

#### MOVLPD (128-bit Legacy SSE load)

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

**VMOVLPD (VEX.128 encoded load)**

DEST[63:0]  $\leftarrow$  SRC2[63:0]  
DEST[127:64]  $\leftarrow$  SRC1[127:64]  
DEST[VLMAX-1:128]  $\leftarrow$  0

**VMOVLPD (store)**

DEST[63:0]  $\leftarrow$  SRC[63:0]

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

MOVLPD:       `__m128d _mm_loadl_pd ( __m128d a, double *p)`  
MOVLPD:       `void _mm_storel_pd (double *p, __m128d a)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5; additionally

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

## MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 12 /r MOVLPS <i>xmm, m64</i>	RM	V/V	SSE	Move two packed single-precision floating-point values from <i>m64</i> to low quadword of <i>xmm</i> .
OF 13 /r MOVLPS <i>m64, xmm</i>	MR	V/V	SSE	Move two packed single-precision floating-point values from low quadword of <i>xmm</i> to <i>m64</i> .
VEX.NDS.128.0F.WIG 12 /r VMOVLPS <i>xmm2, xmm1, m64</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from <i>m64</i> and the high quadword of <i>xmm1</i> .
VEX.128.0F.WIG 13/r VMOVLPS <i>m64, xmm1</i>	MR	V/V	AVX	Move two packed single-precision floating-point values from low quadword of <i>xmm1</i> to <i>m64</i> .

### Instruction Operand Encoding

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

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

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

#### VEX.128 encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (third operand), merges them with the upper 64-bits of the first source XMM register (second operand), and stores them in the low 128-bits of the destination XMM register (first operand). The upper 128-bits of the destination YMM register are zeroed.

#### 128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing OF 13 store. For VMOVLPS (store) (VEX.128.0F 13 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

## Operation

### **MOVLP(S) (128-bit Legacy SSE load)**

DEST[63:0]  $\leftarrow$  SRC[63:0]  
 DEST[VLMAX-1:64] (Unmodified)

### **VMOVLP(S) (VEX.128 encoded load)**

DEST[63:0]  $\leftarrow$  SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VMOVLP(S) (store)**

DEST[63:0]  $\leftarrow$  SRC[63:0]

## Intel C/C++ Compiler Intrinsic Equivalent

MOVLP(S):       `__m128 _mm_loadl_pi ( __m128 a, __m64 *p)`  
 MOVLP(S):       `void _mm_storel_pi ( __m64 *p, __m128 a)`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 5; additionally

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

## MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 50 /r MOVMSKPD <i>reg</i> , <i>xmm</i>		RM	V/V	SSE2 Extract 2-bit sign mask from <i>xmm</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are filled with zeros.
VEX.128.66.0F.WIG 50 /r VMOVMSKPD <i>reg</i> , <i>xmm2</i>		RM	V/V	AVX Extract 2-bit sign mask from <i>xmm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
VEX.256.66.0F.WIG 50 /r VMOVMSKPD <i>reg</i> , <i>ymm2</i>		RM	V/V	AVX Extract 4-bit sign mask from <i>ymm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

### Instruction Operand Encoding

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

### 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)

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 7; 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
OF 50 /r MOVMSKPS <i>reg, xmm</i>	RM	V/V	SSE	Extract 4-bit sign mask from <i>xmm</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are filled with zeros.
VEX.128.0F.WIG 50 /r VMOVMSKPS <i>reg, xmm2</i>	RM	V/V	AVX	Extract 4-bit sign mask from <i>xmm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
VEX.256.0F.WIG 50 /r VMOVMSKPS <i>reg, ymm2</i>	RM	V/V	AVX	Extract 8-bit sign mask from <i>ymm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

### Instruction Operand Encoding<sup>1</sup>

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

### 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 Exceptions Type 7; additionally

#UD                    If VEX.vvvv != 1111B.

## MOVNTDQA – Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA <i>xmm1, m128</i>	RM	V/V	SSE4_1	Move double quadword from <i>m128</i> to <i>xmm</i> using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA <i>xmm1, m128</i>	RM	V/V	AVX	Move double quadword from <i>m128</i> to <i>xmm</i> using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA <i>ymm1, m256</i>	RM	V/V	AVX2	Move 256-bit data from <i>m256</i> to <i>ymm</i> using non-temporal hint if WC memory type.

### Instruction Operand Encoding

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

### Description

(V)MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint. A processor implementation may make use of the non-temporal hint associated with this instruction if the memory source is WC (write combining) memory type. An implementation may also make use of the non-temporal hint associated with this instruction if the memory source is WB (write back) memory type.

A processor's implementation of the non-temporal hint does not override the effective memory type semantics, 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. Another implementation of the hint for WC memory type may optimize data transfer throughput of WC reads. A third implementation may optimize cache reads generated by (V)MOVNTDQA on WB memory type to reduce cache evictions.

### WC Streaming Load Hint

For WC memory type in particular, the processor never appears to read the data into the cache hierarchy. Instead, 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 (V)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 memory currently residing in 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 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 Chapter 11, "Memory Cache Control" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Because the WC protocol uses a weakly-ordered memory consistency model, an MFENCE or locked instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might reference the same WC memory locations or in order to synchronize reads of a processor with writes by other agents in the system.

Because of the speculative nature of fetching due to MOVNTDQA, Streaming loads must not be used to reference memory addresses that are mapped to I/O devices having side effects or when reads to these devices are destruc-

tive. For additional information on MOVNTDQA usages, see Section 12.10.3 in Chapter 12, “Programming with SSE3, SSSE3 and SSE4” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

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.

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

## Operation

### MOVNTDQA (128bit- Legacy SSE form)

DEST  $\leftarrow$  SRC

DEST[VLMAX-1:128] (Unmodified)

### VMOVNTDQA (VEX.128 encoded form)

DEST  $\leftarrow$  SRC

DEST[VLMAX-1:128]  $\leftarrow$  0

### VMOVNTDQA (VEX.256 encoded form)

DEST[255:0]  $\leftarrow$  SRC[255:0]

## Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVNTDQA: `__m128i _mm_stream_load_si128 (__m128i *p);`

VMOVNTDQA: `__m256i _mm256_stream_load_si256 (const __m256i *p);`

## Flags Affected

None

## Other Exceptions

See Exceptions Type 1.SSE4.1; additionally

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

## MOVNTDQ—Store Double Quadword Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ <i>m128, xmm</i>	MR	V/V	SSE2	Move double quadword from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ <i>m128, xmm1</i>	MR	V/V	AVX	Move packed integer values in <i>xmm1</i> to <i>m128</i> using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ <i>m256, ymm1</i>	MR	V/V	AVX	Move packed integer values in <i>ymm1</i> to <i>m256</i> using non-temporal hint.

### Instruction Operand Encoding<sup>1</sup>

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

### Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain integer data (packed bytes, words, doublewords, or quadwords). The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncachable (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 MOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

### Operation

DEST ← SRC;

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

```
MOVNTDQ: void _mm_stream_si128( __m128i *p, __m128i a);
VMOVNTDQ: void _mm256_stream_si256 ( __m256i * p, __m256i a);
```

### SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

**Other Exceptions**

See Exceptions Type 1.SSE2; additionally  
#UD                    If VEX.vvvv != 1111B.

## MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF C3 /r	MOVNTI <i>m32, r32</i>	MR	Valid	Valid	Move doubleword from <i>r32</i> to <i>m32</i> using non-temporal hint.
REX.W + OF C3 /r	MOVNTI <i>m64, r64</i>	MR	Valid	N.E.	Move quadword from <i>r64</i> to <i>m64</i> 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)	NA	NA

### 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[bit 26] = 0. 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.
- #UD            If CPUID.01H:EDX.SSE2[bit 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[bit 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	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD <i>m128</i> , <i>xmm</i>	MR	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD <i>m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move packed double-precision values in <i>xmm1</i> to <i>m128</i> using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD <i>m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move packed double-precision values in <i>ymm1</i> to <i>m256</i> using non-temporal hint.

### Instruction Operand Encoding<sup>1</sup>

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

### Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *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 MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

### Operation

DEST  $\leftarrow$  SRC;

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

MOVNTPD:	void _mm_stream_pd(double *p, __m128d a)
VMOVNTPD:	void _mm256_stream_pd (double * p, __m256d a);

### SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

**Other Exceptions**

See Exceptions Type 1.SSE2; additionally

#UD                    If VEX.vvvv != 1111B.

## MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 2B /r MOVNTPS <i>m128</i> , <i>xmm</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.OF.WIG 2B /r VMOVNTPS <i>m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move packed single-precision values <i>xmm1</i> to <i>mem</i> using non-temporal hint.
VEX.256.OF.WIG 2B /r VMOVNTPS <i>m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move packed single-precision values <i>ymm1</i> to <i>mem</i> using non-temporal hint.

### Instruction Operand Encoding<sup>1</sup>

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

### Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *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 MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

DEST ← SRC;

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

MOVNTDQ: void \_mm\_stream\_ps(float \* p, \_\_m128 a)  
 VMOVNTPS: void \_mm256\_stream\_ps (float \* p, \_\_m256 a);

### SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

**Other Exceptions**

See Exceptions Type 1.SSE; additionally

#UD                    If VEX.vvvv != 1111B.

## MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF E7 /r	MOVNTQ <i>m64, mm</i>	MR	Valid	Valid	Move quadword from <i>mm</i> to <i>m64</i> 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)	NA	NA

### 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 Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**MOVQ—Move Quadword**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 6F /r	RM	V/V	MMX	Move quadword from <i>mm/m64</i> to <i>mm</i> .
MOVQ <i>mm, mm/m64</i>				
OF 7F /r	MR	V/V	MMX	Move quadword from <i>mm</i> to <i>mm/m64</i> .
MOVQ <i>mm/m64, mm</i>				
F3 OF 7E /r	RM	V/V	SSE2	Move quadword from <i>xmm2/mem64</i> to <i>xmm1</i> .
MOVQ <i>xmm1, xmm2/m64</i>				
VEX.128.F3.0F.WIG 7E /r	RM	V/V	AVX	Move quadword from <i>xmm2</i> to <i>xmm1</i> .
VMOVQ <i>xmm1, xmm2</i>				
VEX.128.F3.0F.WIG 7E /r	RM	V/V	AVX	Load quadword from <i>m64</i> to <i>xmm1</i> .
VMOVQ <i>xmm1, m64</i>				
66 OF D6 /r	MR	V/V	SSE2	Move quadword from <i>xmm1</i> to <i>xmm2/mem64</i> .
MOVQ <i>xmm2/m64, xmm1</i>				
VEX.128.66.0F.WIG D6 /r	MR	V/V	AVX	Move quadword from <i>xmm2</i> register to <i>xmm1/m64</i> .
VMOVQ <i>xmm1/m64, xmm2</i>				

**Instruction Operand Encoding**

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

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

Note: In VEX.128.66.0F D6 instruction version, VEX.vvvv and VEX.L=1 are reserved and the former must be 1111b otherwise instructions will #UD.

Note: In VEX.128.F3.0F 7E version, VEX.vvvv and VEX.L=1 are reserved and the former must be 1111b, otherwise instructions will #UD.

**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:

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

MOVQ instruction when source operand is memory location and destination operand is XMM register:

$\text{DEST}[63:0] \leftarrow \text{SRC};$   
 $\text{DEST}[127:64] \leftarrow 0000000000000000H;$

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination:

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[VLMAX-1:64] \leftarrow 0$

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination:

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[VLMAX-1:64] \leftarrow 0$

VMOVQ (7E) with memory source:

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$   
 $\text{DEST}[VLMAX-1:64] \leftarrow 0$

VMOVQ (D6) with memory dest:

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

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

MOVQ:  $\text{m128i } \text{_mm\_mov\_epi64}(\text{__m128i } \text{a})$

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

**MOVQ2DQ—Move Quadword from MMX Technology to XMM Register**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 OF D6 /r	MOVQ2DQ <i>xmm, mm</i>	RM	Valid	Valid	Move quadword from <i>mmx</i> to low quadword of <i>xmm</i> .

**Instruction Operand Encoding**

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

**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\_pi64 ( \_\_m64 a)

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM	If CRO.TS[bit 3] = 1.
#UD	If CRO.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 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 <i>m8, m8</i>	NP	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 <i>m16, m16</i>	NP	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 <i>m32, m32</i>	NP	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 <i>m64, m64</i>	NP	Valid	N.E.	Move qword from address (R E)SI to (R E)DI.
A4	MOVSB	NP	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	NP	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	NP	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	NP	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
NP	NA	NA	NA	NA

### 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 MOVSQ instructions can be preceded by the REP prefix (see "REP/REPE/REPZ/REPNE/REPNZ—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) 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  $\leftarrow$  SRC;

Non-64-bit Mode:

```

IF (Byte move)
  THEN IF DF = 0
    THEN
      (E)SI  $\leftarrow$  (E)SI + 1;
      (E)DI  $\leftarrow$  (E)DI + 1;
    ELSE
      (E)SI  $\leftarrow$  (E)SI - 1;
      (E)DI  $\leftarrow$  (E)DI - 1;
    FI;
  ELSE IF (Word move)
    THEN IF DF = 0
      (E)SI  $\leftarrow$  (E)SI + 2;
      (E)DI  $\leftarrow$  (E)DI + 2;
    FI;
  ELSE
    (E)SI  $\leftarrow$  (E)SI - 2;
    (E)DI  $\leftarrow$  (E)DI - 2;
  FI;
ELSE IF (Doubleword move)
  THEN IF DF = 0
    (E)SI  $\leftarrow$  (E)SI + 4;
    (E)DI  $\leftarrow$  (E)DI + 4;
  FI;
  ELSE
    (E)SI  $\leftarrow$  (E)SI - 4;
    (E)DI  $\leftarrow$  (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 Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description	
F2 OF 10 /r		RM	V/V	SSE2	Move scalar double-precision floating-point value from <i>xmm2/m64</i> to <i>xmm1</i> register.
MOVSD <i>xmm1, xmm2/m64</i>					
VEX.NDS.LIG.F2.0F.WIG 10 /r		RVM	V/V	AVX	Merge scalar double-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VMOVSD <i>xmm1, xmm2, xmm3</i>					
VEX.LIG.F2.0F.WIG 10 /r		XM	V/V	AVX	Load scalar double-precision floating-point value from <i>m64</i> to <i>xmm1</i> register.
VMOVSD <i>xmm1, m64</i>					
F2 OF 11 /r		MR	V/V	SSE2	Move scalar double-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m64</i> .
MOVSD <i>xmm2/m64, xmm1</i>					
VEX.NDS.LIG.F2.0F.WIG 11 /r		MVR	V/V	AVX	Merge scalar double-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> registers to <i>xmm1</i> .
VMOVSD <i>xmm1, xmm2, xmm3</i>					
VEX.LIG.F2.0F.WIG 11 /r		MR	V/V	AVX	Move scalar double-precision floating-point value from <i>xmm1</i> register to <i>m64</i> .
VMOVSD <i>m64, xmm1</i>					

### Instruction Operand Encoding

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

### Description

MOVSD 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.

For non-VEX encoded instruction syntax and when the source and destination operands are XMM registers, the high quadword of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high quadword of the destination operand is cleared to all 0s.

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

Note: For the “VMOVSD *m64, xmm1*” (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b, otherwise instruction will #UD.

Note: For the “VMOVSD *xmm1, m64*” (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

VEX encoded instruction syntax supports two source operands and a destination operand if ModR/M.mod field is 11B. VEX.vvvv is used to encode the first source operand (the second operand). The low 128 bits of the destination operand stores the result of merging the low quadword of the second source operand with the quad word in bits 127:64 of the first source operand. The upper bits of the destination operand are cleared.

## Operation

**MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)**

DEST[63:0] ← SRC[63:0]

DEST[VLMAX-1:64] (Unmodified)

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

**VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)**

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

**VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)**

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

**VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)**

DEST[63:0] ← SRC[63:0]

DEST[VLMAX-1:64] ← 0

## Intel C/C++ Compiler Intrinsic Equivalent

MOVSD:     `__m128d _mm_load_sd (double *p)`

MOVSD:     `void _mm_store_sd (double *p, __m128d a)`

MOVSD:     `__m128d _mm_store_sd (__m128d a, __m128d b)`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 5; additionally

#UD                 If VEX.vvvv != 1111B.

## MOVSHDUP—Move Packed Single-FP High and Duplicate

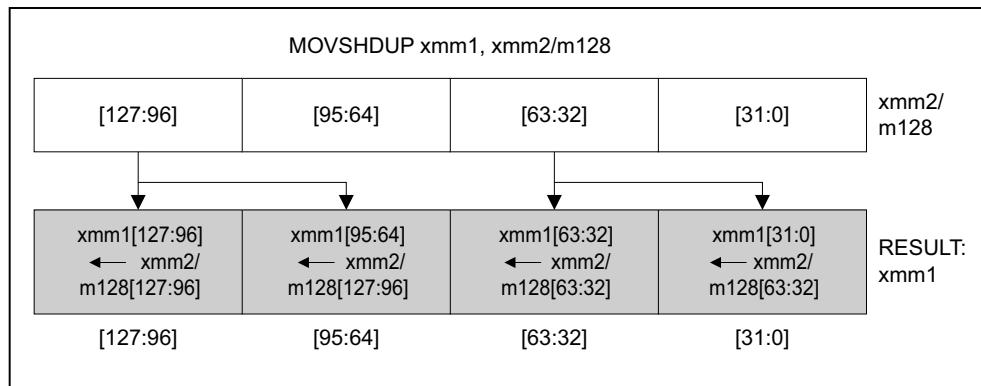
Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 16 /r MOVSHDUP <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Move two single-precision floating-point values from the higher 32-bit operand of each qword in <i>xmm2/m128</i> to <i>xmm1</i> and duplicate each 32-bit operand to the lower 32-bits of each qword.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move odd index single-precision floating-point values from <i>xmm2/mem</i> and duplicate each element into <i>xmm1</i> .
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move odd index single-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

### Instruction Operand Encoding

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

### Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded and the single-precision elements in positions 1 and 3 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register. See Figure 3-25.



OM15998

**Figure 3-25. MOVSHDUP—Move Packed Single-FP High and Duplicate**

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

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

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

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

## Operation

### **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[VLMAX-1:128] (Unmodified)

### **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[VLMAX-1:128] ← 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]

## Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVSHDUP:     `__m128 _mm_movehdup_ps(__m128 a)`  
 VMOVSHDUP:     `__m256 _mm256_movehdup_ps (__m256 a);`

## Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

## Numeric Exceptions

None

## Other Exceptions

See Exceptions Type 2.

## MOVSLDUP—Move Packed Single-FP Low and Duplicate

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 12 /r MOVSLDUP <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Move two single-precision floating-point values from the lower 32-bit operand of each qword in <i>xmm2/m128</i> to <i>xmm1</i> and duplicate each 32-bit operand to the higher 32-bits of each qword.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move even index single-precision floating-point values from <i>xmm2/mem</i> and duplicate each element into <i>xmm1</i> .
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move even index single-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

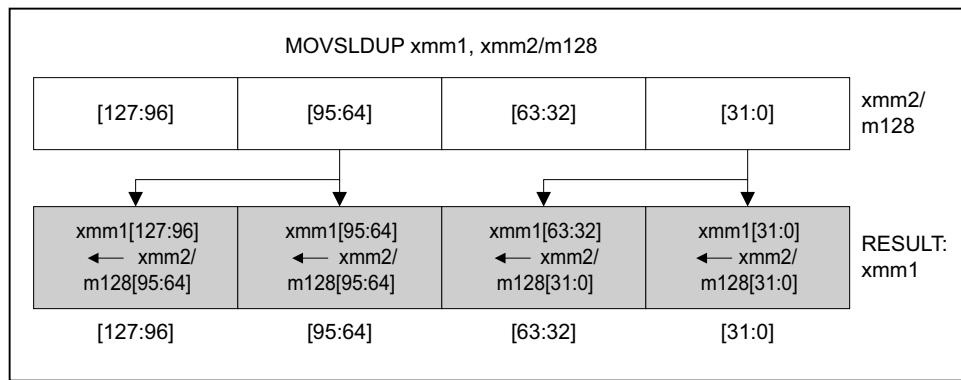
### Instruction Operand Encoding

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

### Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded and the single-precision elements in positions 0 and 2 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register.

See Figure 3-26.



OM15999

Figure 3-26. MOVSLDUP—Move Packed Single-FP Low and Duplicate

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

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

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

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

## Operation

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

### **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[VLMAX-1:128] ← 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]
```

## Intel C/C++ Compiler Intrinsic Equivalent

```
(V)MOVSLDUP:     __m128 _mm_moveldup_ps(__m128 a)
VMOVSLDUP:      __m256 _mm256_moveldup_ps (__m256 a);
```

## Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

## Numeric Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.vvvv != 1111B.

## MOVSS—Move Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 10 /r MOVSS <i>xmm1, xmm2/m32</i>	RM	V/V	SSE	Move scalar single-precision floating-point value from <i>xmm2/m32</i> to <i>xmm1</i> register.
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS <i>xmm1, xmm2, xmm3</i>	RVM	V/V	AVX	Merge scalar single-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS <i>xmm1, m32</i>	XM	V/V	AVX	Load scalar single-precision floating-point value from <i>m32</i> to <i>xmm1</i> register.
F3 OF 11 /r MOVSS <i>xmm2/m32, xmm</i>	MR	V/V	SSE	Move scalar single-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m32</i> .
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS <i>xmm1, xmm2, xmm3</i>	MVR	V/V	AVX	Move scalar single-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS <i>m32, xmm1</i>	MR	V/V	AVX	Move scalar single-precision floating-point value from <i>xmm1</i> register to <i>m32</i> .

### Instruction Operand Encoding

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

### Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

For non-VEX encoded syntax and when the source and destination operands are XMM registers, the high doublewords of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high doublewords of the destination operand is cleared to all Os.

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

VEX encoded instruction syntax supports two source operands and a destination operand if ModR/M.mod field is 11B. VEX.vvvv is used to encode the first source operand (the second operand). The low 128 bits of the destination operand stores the result of merging the low dword of the second source operand with three dwords in bits 127:32 of the first source operand. The upper bits of the destination operand are cleared.

Note: For the “VMOVSS *m32, xmm1*” (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the “VMOVSS *xmm1, m32*” (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

## Operation

**MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)**

DEST[31:0]  $\leftarrow$  SRC[31:0]

DEST[VLMAX-1:32] (Unmodified)

**MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)**

DEST[31:0]  $\leftarrow$  SRC[31:0]

**MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)**

DEST[31:0]  $\leftarrow$  SRC[31:0]

DEST[127:32]  $\leftarrow$  0

DEST[VLMAX-1:128] (Unmodified)

**VMOVSS (VEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)**

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

DEST[127:32]  $\leftarrow$  SRC1[127:32]

DEST[VLMAX-1:128]  $\leftarrow$  0

**VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)**

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

DEST[127:32]  $\leftarrow$  SRC1[127:32]

DEST[VLMAX-1:128]  $\leftarrow$  0

**VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)**

DEST[31:0]  $\leftarrow$  SRC[31:0]

DEST[VLMAX-1:32]  $\leftarrow$  0

## Intel C/C++ Compiler Intrinsic Equivalent

MOVSS:     `__m128 _mm_load_ss(float * p)`

MOVSS:     `void _mm_store_ss(float * p, __m128 a)`

MOVSS:     `__m128 _mm_move_ss(__m128 a, __m128 b)`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 5; additionally

#UD                 If VEX.vvvv != 1111B.

## MOVsx/MOVsxD—Move with Sign-Extension

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF BE /r	MOVsx r16, r/m8	RM	Valid	Valid	Move byte to word with sign-extension.
OF BE /r	MOVsx r32, r/m8	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX + OF BE /r	MOVsx r64, r/m8*	RM	Valid	N.E.	Move byte to quadword with sign-extension.
OF BF /r	MOVsx r32, r/m16	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + OF BF /r	MOVsx r64, r/m16	RM	Valid	N.E.	Move word to quadword with sign-extension.
REX.W** + 63 /r	MOVsxD r64, r/m32	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

### 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.

\*\* 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)	NA	NA

### 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	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD <i>xmm1, xmm2/m128</i>		RM	V/V	SSE2 Move packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 10 /r VMOVUPD <i>xmm1, xmm2/m128</i>		RM	V/V	AVX Move unaligned packed double-precision floating-point from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.256.66.0F.WIG 10 /r VMOVUPD <i>ymm1, ymm2/m256</i>		RM	V/V	AVX Move unaligned packed double-precision floating-point from <i>ymm2/mem</i> to <i>ymm1</i> .
66 0F 11 /r MOVUPD <i>xmm2/m128, xmm</i>		MR	V/V	SSE2 Move packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 11 /r VMOVUPD <i>xmm2/m128, xmm1</i>		MR	V/V	AVX Move unaligned packed double-precision floating-point from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 11 /r VMOVUPD <i>ymm2/m256, ymm1</i>		MR	V/V	AVX Move unaligned packed double-precision floating-point from <i>ymm1</i> to <i>ymm2/mem</i> .

### Instruction Operand Encoding

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

### Description

#### 128-bit versions:

Moves a double quadword containing two 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, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers.

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

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

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.<sup>1</sup>

To move double-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPD instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

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

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.

1. 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 operand is not aligned on an 8-byte boundary.

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

## Operation

### **MOVUPD (128-bit load and register-copy form Legacy SSE version)**

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128] (Unmodified)

### **(V)MOVUPD (128-bit store form)**

DEST[127:0]  $\leftarrow$  SRC[127:0]

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

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VMOVUPD (VEX.256 encoded version)**

DEST[255:0]  $\leftarrow$  SRC[255:0]

## Intel C/C++ Compiler Intrinsic Equivalent

```
MOVUPD:    __m128 _mm_loadu_pd(double * p)
MOVUPD:    void _mm_storeu_pd(double *p, __m128 a)
VMOVUPD:   __m256d _mm256_loadu_pd (__m256d * p);
VMOVUPD:   _mm256_storeu_pd(__m256d *p, __m256d a);
```

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally

#UD                    If VEX.vvvv != 1111B.

## MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 10 /r MOVUPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Move packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.128.0F.WIG 10 /r VMOVUPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move unaligned packed single-precision floating-point from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.256.0F.WIG 10 /r VMOVUPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move unaligned packed single-precision floating-point from <i>ymm2/mem</i> to <i>ymm1</i> .
OF 11 /r MOVUPS <i>xmm2/m128, xmm1</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.0F.WIG 11 /r VMOVUPS <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move unaligned packed single-precision floating-point from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.0F.WIG 11 /r VMOVUPS <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move unaligned packed single-precision floating-point from <i>ymm1</i> to <i>ymm2/mem</i> .

### Instruction Operand Encoding

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

### Description

128-bit versions: Moves a double quadword containing four 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, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers.

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

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

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.<sup>1</sup>

To move packed single-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPS instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

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

VEX.256 encoded version: Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

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

1. 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 operand is not aligned on an 8-byte boundary.

## Operation

### **MOVUPS (128-bit load and register-copy form Legacy SSE version)**

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128] (Unmodified)

### **(V)MOVUPS (128-bit store form)**

DEST[127:0]  $\leftarrow$  SRC[127:0]

### **VMOVUPS (VEX.128 encoded load-form)**

DEST[127:0]  $\leftarrow$  SRC[127:0]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VMOVUPS (VEX.256 encoded version)**

DEST[255:0]  $\leftarrow$  SRC[255:0]

## Intel C/C++ Compiler Intrinsic Equivalent

```
MOVUPS:    __m128 _mm_loadu_ps(double * p)
MOVUPS:    void _mm_storeu_ps(double *p, __m128 a)
VMOVUPS:   __m256 _mm256_loadu_ps (__m256 * p);
VMOVUPS:   _mm256_storeu_ps(__m256 *p, __m256 a);
```

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally

#UD                    If 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	RM	Valid	Valid	Move byte to word with zero-extension.
0F B6 /r	MOVZX r32, r/m8	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + 0F B6 /r	MOVZX r64, r/m8*	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:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

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

### 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 <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and writes the results in <i>xmm1</i> . Starting offsets within <i>xmm1</i> and <i>xmm2/m128</i> are determined by <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 42 /r ib VMPSADBW <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and writes the results in <i>xmm1</i> . Starting offsets within <i>xmm2</i> and <i>xmm3/m128</i> are determined by <i>imm8</i> .
VEX.NDS.256.66.0F3A.WIG 42 /r ib VMPSADBW <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm2</i> and <i>ymm3/m128</i> and writes the results in <i>ymm1</i> . Starting offsets within <i>ymm2</i> and <i>xmm3/m128</i> are determined by <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

(V)MPSADBW sums the absolute difference of 4 unsigned bytes (block\_2) in the second source operand with sequential groups of 4 unsigned bytes (block\_1) in the first source operand. The immediate byte provides bit fields that specify the initial offset of block\_1 within the first source operand, and the offset of block\_2 within the second source operand. The offset granularity in both source operands are 32 bits. The sum-absolute-difference (SAD) operation is repeated 8 times for (V)MPSADW between the same block\_2 (fixed offset within the second source operand) and a variable block\_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations is written to the respective word in the destination operand.

128-bit Legacy SSE version: Imm8[1:0]\*32 specifies the bit offset of block\_2 within the second source operand. Imm[2]\*32 specifies the initial bit offset of the block\_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (VLMAX-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.

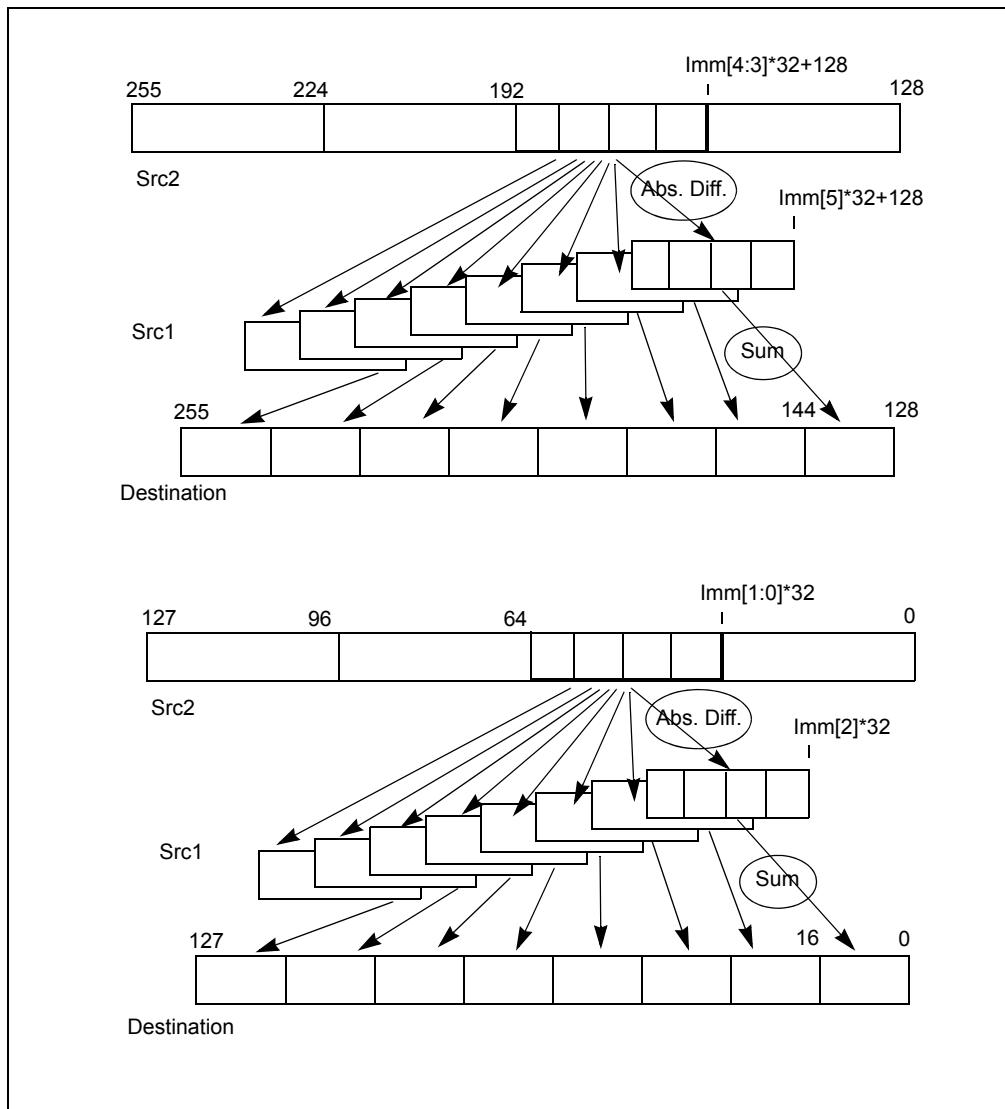


Figure 3-27. VMPSADBW Operation

## Operation

### VMPSADBW (VEX.256 encoded version)

```

SRC2_OFFSET <- imm8[1:0]*32
SRC1_OFFSET <- imm8[2]*32
SRC1_BYTETO <- SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYTE1 <- SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYTE2 <- SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYTE3 <- SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYTE4 <- SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYTE5 <- SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]

```

```

SRC1_BYT6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYT7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYT8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYT9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYT10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]
SRC2_BYT0 ← SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYT1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYT2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYT3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

```

```

TEMPO ← ABS(SRC1_BYT0 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT1 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT2 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT3 - SRC2_BYT3)
DEST[15:0] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT1 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT2 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT3 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT4 - SRC2_BYT3)
DEST[31:16] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT2 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT3 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT4 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT5 - SRC2_BYT3)
DEST[47:32] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT3 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT4 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT5 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT6 - SRC2_BYT3)
DEST[63:48] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT4 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT5 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT6 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT7 - SRC2_BYT3)
DEST[79:64] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT5 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT6 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT7 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT8 - SRC2_BYT3)
DEST[95:80] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT6 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT7 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT8 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT9 - SRC2_BYT3)
DEST[111:96] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT7 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT8 - SRC2_BYT1)

```

## INSTRUCTION SET REFERENCE, A-M

```

TEMP2 ← ABS(SRC1_BYTEn - SRC2_BYTEn)
TEMP3 ← ABS(SRC1_BYTEn+1 - SRC2_BYTEn+1)
DEST[127:112] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

SRC2_OFFSET ← imm8[4:3]*32 + 128
SRC1_OFFSET ← imm8[5]*32 + 128
SRC1_BYTEn ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYTEn+1 ← SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYTEn+2 ← SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYTEn+3 ← SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYTEn+4 ← SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYTEn+5 ← SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
SRC1_BYTEn+6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYTEn+7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYTEn+8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYTEn+9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYTEn+10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]

```

```

SRC2_BYTEn ← SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYTEn+1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYTEn+2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYTEn+3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

```

```

TEMPO ← ABS(SRC1_BYTEn - SRC2_BYTEn)
TEMP1 ← ABS(SRC1_BYTEn+1 - SRC2_BYTEn+1)
TEMP2 ← ABS(SRC1_BYTEn+2 - SRC2_BYTEn+2)
TEMP3 ← ABS(SRC1_BYTEn+3 - SRC2_BYTEn+3)
DEST[143:128] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+4 - SRC2_BYTEn+4)
TEMP1 ← ABS(SRC1_BYTEn+5 - SRC2_BYTEn+5)
TEMP2 ← ABS(SRC1_BYTEn+6 - SRC2_BYTEn+6)
TEMP3 ← ABS(SRC1_BYTEn+7 - SRC2_BYTEn+7)
DEST[159:144] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+8 - SRC2_BYTEn+8)
TEMP1 ← ABS(SRC1_BYTEn+9 - SRC2_BYTEn+9)
TEMP2 ← ABS(SRC1_BYTEn+10 - SRC2_BYTEn+10)
TEMP3 ← ABS(SRC1_BYTEn+11 - SRC2_BYTEn+11)
DEST[175:160] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+12 - SRC2_BYTEn+12)
TEMP1 ← ABS(SRC1_BYTEn+13 - SRC2_BYTEn+13)
TEMP2 ← ABS(SRC1_BYTEn+14 - SRC2_BYTEn+14)
TEMP3 ← ABS(SRC1_BYTEn+15 - SRC2_BYTEn+15)
DEST[191:176] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+16 - SRC2_BYTEn+16)
TEMP1 ← ABS(SRC1_BYTEn+17 - SRC2_BYTEn+17)
TEMP2 ← ABS(SRC1_BYTEn+18 - SRC2_BYTEn+18)
TEMP3 ← ABS(SRC1_BYTEn+19 - SRC2_BYTEn+19)
DEST[207:192] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+20 - SRC2_BYTEn+20)

```

```

TEMP1 ← ABS(SRC1_BYT6 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT7 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT8 - SRC2_BYT3)
DEST[223:208] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT6 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT7 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT8 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT9 - SRC2_BYT3)
DEST[239:224] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT7 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT8 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT9 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT10 - SRC2_BYT3)
DEST[255:240] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

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

```

SRC2_OFFSET ← imm8[1:0]*32
SRC1_OFFSET ← imm8[2]*32
SRC1_BYT0 ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYT1 ← SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYT2 ← SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYT3 ← SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYT4 ← SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYT5 ← SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
SRC1_BYT6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYT7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYT8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYT9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYT10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]

```

```

SRC2_BYT0 ← SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYT1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYT2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYT3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

```

```

TEMPO ← ABS(SRC1_BYT0 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT1 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT2 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT3 - SRC2_BYT3)
DEST[15:0] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT1 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT2 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT3 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT4 - SRC2_BYT3)
DEST[31:16] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT2 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT3 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT4 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT5 - SRC2_BYT3)
DEST[47:32] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3
DEST[VLMAX-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]

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← 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] ← TEMPO + TEMP1 + TEMP2 + TEMP3
DEST[VLMAX-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 Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## MUL—Unsigned Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /4	MUL r/m8	M	Valid	Valid	Unsigned multiply ( $AX \leftarrow AL * r/m8$ ).
REX + F6 /4	MUL r/m8*	M	Valid	N.E.	Unsigned multiply ( $AX \leftarrow AL * r/m8$ ).
F7 /4	MUL r/m16	M	Valid	Valid	Unsigned multiply ( $DX:AX \leftarrow AX * r/m16$ ).
F7 /4	MUL r/m32	M	Valid	Valid	Unsigned multiply ( $EDX:EAX \leftarrow EAX * r/m32$ ).
REX.W + F7 /4	MUL r/m64	M	Valid	N.E.	Unsigned multiply ( $RDX:RAX \leftarrow RAX * r/m64$ ).

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### 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 3-66.

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 3-66. 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; Fl;
      ELSE (* OperandSize = 64 *)
        RDX:RAX ← RAX * SRC;
    Fl;
Fl;

```

## 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	CPUID Feature Flag	Description
66 0F 59 /r MULPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Multiply packed double-precision floating-point values in <i>xmm2/m128</i> by <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 59 /r VMULPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Multiply packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG 59 /r VMULPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Multiply packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

### Instruction Operand Encoding

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

### Description

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

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

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

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

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

### Operation

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

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

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

#### 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]
```

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

MULPD:       `__m128d _mm_mmul_pd (m128d a, m128d b)`

VMULPD:      `__m256d _mm256_mmul_pd (__m256d a, __m256d b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2

## MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 59 /r MULPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Multiply packed single-precision floating-point values in <i>xmm2/mem</i> by <i>xmm1</i> .
VEX.NDS.128.0F.WIG 59 /r VMULPS <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Multiply packed single-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.0F.WIG 59 /r VMULPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Multiply packed single-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

### Instruction Operand Encoding

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

### Description

Performs a SIMD multiply of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

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

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

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

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

### Operation

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

#### 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[VLMAX-1:128] ← 0
```

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

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] * \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] * \text{SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] * \text{SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] * \text{SRC2}[255:224].$

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

MULPS:       `__m128 _mm_mul_ps(__m128 a, __m128 b)`  
 VMULPS:      `__m256 _mm256_mul_ps (__m256 a, __m256 b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2

## MULSD—Multiply Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 59 /r MULSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Multiply the low double-precision floating-point value in <i>xmm2/mem64</i> by low double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.OF.WIG 59/r VMULSD <i>xmm1,xmm2, xmm3/m64</i>	RVM	V/V	AVX	Multiply the low double-precision floating-point value in <i>xmm3/mem64</i> by low double precision floating-point value in <i>xmm2</i> .

### Instruction Operand Encoding

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

### Description

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

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

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

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

### Operation

#### MULSD (128-bit Legacy SSE version)

$DEST[63:0] \leftarrow DEST[63:0] * SRC[63:0]$   
 $DEST[VLMAX-1:64] \text{ (Unmodified)}$

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

$DEST[63:0] \leftarrow SRC1[63:0] * SRC2[63:0]$   
 $DEST[127:64] \leftarrow SRC1[127:64]$   
 $DEST[VLMAX-1:128] \leftarrow 0$

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

MULSD:  $\_m128d\_mm\_mul\_sd(m128d\ a, m128d\ b)$

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Other Exceptions

See Exceptions Type 3

**MULSS—Multiply Scalar Single-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 59 /r MULSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Multiply the low single-precision floating-point value in <i>xmm2/mem</i> by the low single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 59 /r VMULSS <i>xmm1,xmm2, xmm3/m32</i>	RVM	V/V	AVX	Multiply the low single-precision floating-point value in <i>xmm3/mem</i> by the low single-precision floating-point value in <i>xmm2</i> .

**Instruction Operand Encoding**

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

**Description**

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

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

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

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

**Operation****MULSS (128-bit Legacy SSE version)**

DEST[31:0]  $\leftarrow$  DEST[31:0] \* SRC[31:0]  
DEST[VLMAX-1:32] (Unmodified)

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

DEST[31:0]  $\leftarrow$  SRC1[31:0] \* SRC2[31:0]  
DEST[127:32]  $\leftarrow$  SRC1[127:32]  
DEST[VLMAX-1:128]  $\leftarrow$  0

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

MULSS: `__m128 _mm_mul_ss(__m128 a, __m128 b)`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 3

## MULX – Unsigned Multiply Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.F2.0F38.W0 F6 /r MULX r32a, r32b, r/m32	RVM	V/V	BMI2	Unsigned multiply of <i>r/m32</i> with EDX without affecting arithmetic flags.
VEX.NDD.LZ.F2.0F38.W1 F6 /r MULX r64a, r64b, r/m64	RVM	V/N.E.	BMI2	Unsigned multiply of <i>r/m64</i> 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
```

### Flags Affected

None

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

Auto-generated from high-level language when possible.

```
unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);
unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);
```

### SIMD Floating-Point Exceptions

None

**Other Exceptions**

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

#UD                  If VEX.W = 1.

## MWAIT—Monitor Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 C9	MWAIT	NP	Valid	Valid	A hint that allow 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
NP	NA	NA	NA	NA

### Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT 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 MWAIT clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

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

ECX specifies optional 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 (see below). Software can use the CPUID instruction to determine the extensions and hints supported by the processor.

### MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a 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[bit 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. Support for MWAIT extensions for power management is indicated by CPUID.05H:ECX[bit 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 3-67 describes the meaning of ECX and EAX registers for MWAIT extensions.

**Table 3-67. 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[bit 1] = 1.
31: 1	Reserved

**Table 3-68. 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 14, “Power and Thermal Management,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## 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”)){
    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

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 un-noticed. 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[31:1] ≠ 0. If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0. If current privilege level is not 0.

## Real Address Mode Exceptions

#GP	If ECX[31:1] ≠ 0. If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

## Virtual 8086 Mode Exceptions

#UD	The MWAIT instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).
-----	--

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#GP(0)	If RCX[63:1] ≠ 0. If RCX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If the current privilege level is not 0. If CPUID.01H:ECX.MONITOR[bit 3] = 0.

## INSTRUCTION SET REFERENCE, A-M

# CHAPTER 4 INSTRUCTION SET REFERENCE, N-Z

## 4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM

The notations introduced in this section are referenced in the reference pages of PCMPESTRI, PCMPESTRM, 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 bit[3:2].

See Section 4.1.6 for a description of the overrideIfDataInvalid() function used in Table 4-3.

**Table 4-3. Aggregation Operation**

Mode	Pseudocode
Equal any (find characters from a set)	UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For j = 0 to UpperBound, j++ For i = 0 to UpperBound, i++ IntRes1[j] OR= overrideIfDataInvalid(BoolRes[j,i])
Ranges (find characters from ranges)	UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For j = 0 to UpperBound, j++ For i = 0 to UpperBound, i+=2 IntRes1[j] OR= (overrideIfDataInvalid(BoolRes[j,i]) AND overrideIfDataInvalid(BoolRes[j,i+1]))
Equal each (string compare)	UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For i = 0 to UpperBound, i++ IntRes1[i] = overrideIfDataInvalid(BoolRes[i,i])
Equal ordered (substring search)	UpperBound = imm8[0] ? 7 : 15; IntRes1 = imm8[0] ? 0xFF : 0xFFFF For j = 0 to UpperBound, j++ For i = 0 to UpperBound-j, k=j to UpperBound, k++, i++ IntRes1[j] AND= overrideIfDataInvalid(BoolRes[k,i])

#### 4.1.4 Polarity

IntRes1 may then be further modified by performing a 1's complement, according to the value of the Imm8 Control Byte bit[4]. Optionally, a mask may be used such that only those IntRes1 bits which correspond to "valid" reg/mem input elements are complemented (note that the definition of a valid input element is dependant on the specific opcode and is defined in each opcode's description). The result of the possible negation is referred to as IntRes2.

**Table 4-4. Polarity**

Imm8[5:4]	Operation	Description
00b	Positive Polarity (+)	IntRes2 = IntRes1
01b	Negative Polarity (-)	IntRes2 = -1 XOR IntRes1
10b	Masked (+)	IntRes2 = IntRes1
11b	Masked (-)	IntRes2[i] = IntRes1[i] if reg/mem[i] invalid, else = ~IntRes1[i]

#### 4.1.5 Output Selection

**Table 4-5. Ouput 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

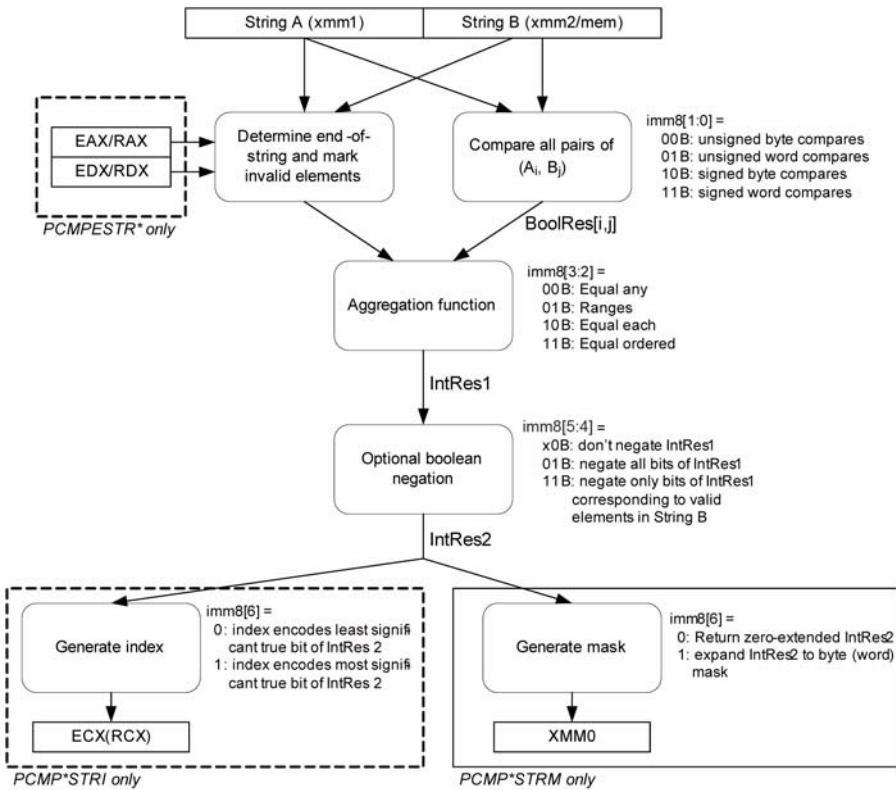


Figure 4-1. Operation of PCMPSTRx and PCMPESTRx

## 4.2 INSTRUCTIONS (N-Z)

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

## NEG—Two's Complement Negation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /3	NEG r/m8	M	Valid	Valid	Two's complement negate r/m8.
REX + F6 /3	NEG r/m8*	M	Valid	N.E.	Two's complement negate r/m8.
F7 /3	NEG r/m16	M	Valid	Valid	Two's complement negate r/m16.
F7 /3	NEG r/m32	M	Valid	Valid	Two's complement negate r/m32.
REX.W + F7 /3	NEG r/m64	M	Valid	N.E.	Two's complement negate r/m64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

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

### Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

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

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

### Operation

```
IF DEST = 0
  THEN CF ← 0;
  ELSE CF ← 1;
FI;
DEST ← [- (DEST)]
```

### Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

### Protected Mode Exceptions

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

### Real-Address Mode Exceptions

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

### Virtual-8086 Mode Exceptions

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

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

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

## NOP—No Operation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
90	NOP	NP	Valid	Valid	One byte no-operation instruction.
OF 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
OF 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

### Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of “no operation” as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

**Table 4-9. Recommended Multi-Byte Sequence of NOP Instruction**

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

## NOT—One's Complement Negation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /2	NOT r/m8	M	Valid	Valid	Reverse each bit of r/m8.
REX + F6 /2	NOT r/m8*	M	Valid	N.E.	Reverse each bit of r/m8.
F7 /2	NOT r/m16	M	Valid	Valid	Reverse each bit of r/m16.
F7 /2	NOT r/m32	M	Valid	Valid	Reverse each bit of r/m32.
REX.W + F7 /2	NOT r/m64	M	Valid	N.E.	Reverse each bit of r/m64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

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

### Description

Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

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

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

### Operation

DEST ← NOT DEST;

### Flags Affected

None.

### Protected Mode Exceptions

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

### Real-Address Mode Exceptions

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

**Virtual-8086 Mode Exceptions**

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

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

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

## OR—Logical Inclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0C <i>ib</i>	OR AL, <i>imm8</i>	I	Valid	Valid	AL OR <i>imm8</i> .
0D <i>iw</i>	OR AX, <i>imm16</i>	I	Valid	Valid	AX OR <i>imm16</i> .
0D <i>id</i>	OR EAX, <i>imm32</i>	I	Valid	Valid	EAX OR <i>imm32</i> .
REX.W + 0D <i>id</i>	OR RAX, <i>imm32</i>	I	Valid	N.E.	RAX OR <i>imm32</i> ( <i>sign-extended</i> ).
80 /1 <i>ib</i>	OR r/m8, <i>imm8</i>	MI	Valid	Valid	r/m8 OR <i>imm8</i> .
REX + 80 /1 <i>ib</i>	OR r/m8*, <i>imm8</i>	MI	Valid	N.E.	r/m8 OR <i>imm8</i> .
81 /1 <i>iw</i>	OR r/m16, <i>imm16</i>	MI	Valid	Valid	r/m16 OR <i>imm16</i> .
81 /1 <i>id</i>	OR r/m32, <i>imm32</i>	MI	Valid	Valid	r/m32 OR <i>imm32</i> .
REX.W + 81 /1 <i>id</i>	OR r/m64, <i>imm32</i>	MI	Valid	N.E.	r/m64 OR <i>imm32</i> ( <i>sign-extended</i> ).
83 /1 <i>ib</i>	OR r/m16, <i>imm8</i>	MI	Valid	Valid	r/m16 OR <i>imm8</i> ( <i>sign-extended</i> ).
83 /1 <i>ib</i>	OR r/m32, <i>imm8</i>	MI	Valid	Valid	r/m32 OR <i>imm8</i> ( <i>sign-extended</i> ).
REX.W + 83 /1 <i>ib</i>	OR r/m64, <i>imm8</i>	MI	Valid	N.E.	r/m64 OR <i>imm8</i> ( <i>sign-extended</i> ).
08 /r	OR r/m8, r8	MR	Valid	Valid	r/m8 OR r8.
REX + 08 /r	OR r/m8*, r8*	MR	Valid	N.E.	r/m8 OR r8.
09 /r	OR r/m16, r16	MR	Valid	Valid	r/m16 OR r16.
09 /r	OR r/m32, r32	MR	Valid	Valid	r/m32 OR r32.
REX.W + 09 /r	OR r/m64, r64	MR	Valid	N.E.	r/m64 OR r64.
0A /r	OR r8, r/m8	RM	Valid	Valid	r8 OR r/m8.
REX + 0A /r	OR r8*, r/m8*	RM	Valid	N.E.	r8 OR r/m8.
0B /r	OR r16, r/m16	RM	Valid	Valid	r16 OR r/m16.
0B /r	OR r32, r/m32	RM	Valid	Valid	r32 OR r/m32.
REX.W + 0B /r	OR r64, r/m64	RM	Valid	N.E.	r64 OR r/m64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

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

### Description

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

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

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

### Operation

DEST ← DEST OR SRC;

### Flags Affected

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

### Protected Mode Exceptions

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

### Real-Address Mode Exceptions

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

### Virtual-8086 Mode Exceptions

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

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

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

## ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56 /r ORPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 56 /r VORPD <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 56 /r VORPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

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

### Description

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

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

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

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

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

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

### Operation

#### ORPD (128-bit Legacy SSE version)

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

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

```
DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

#### VORPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]
```

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

ORPD:       `__m128d _mm_or_pd(__m128d a, __m128d b);`  
VORPD:      `__m256d _mm256_or_pd (__m256d a, __m256d b);`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD           If VEX.L = 1.

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

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

### Instruction Operand Encoding

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

### Description

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

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

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

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

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

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

### Operation

#### ORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

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

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

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

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE OR SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE OR SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE OR SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE OR SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] \text{ BITWISE OR SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] \text{ BITWISE OR SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] \text{ BITWISE OR SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \text{ BITWISE OR SRC2}[255:224].$

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

ORPS:        `__m128 _mm_or_ps (__m128 a, __m128 b);`  
 VORPS:      `__m256 _mm256_or_ps (__m256 a, __m256 b);`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4.

## OUT—Output to Port

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E6 <i>ib</i>	OUT <i>imm8</i> , AL	I	Valid	Valid	Output byte in AL to I/O port address <i>imm8</i> .
E7 <i>ib</i>	OUT <i>imm8</i> , AX	I	Valid	Valid	Output word in AX to I/O port address <i>imm8</i> .
E7 <i>ib</i>	OUT <i>imm8</i> , EAX	I	Valid	Valid	Output doubleword in EAX to I/O port address <i>imm8</i> .
EE	OUT DX, AL	NP	Valid	Valid	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	NP	Valid	Valid	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	NP	Valid	Valid	Output doubleword in EAX to I/O port address in DX.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	imm8	NA	NA	NA
NP	NA	NA	NA	NA

### Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 16, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

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

### IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

## Operation

```

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
    IF (Any I/O Permission Bit for I/O port being accessed = 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
    ELSE (* I/O operation is allowed *)
      DEST ← SRC; (* Writes to selected I/O port *)
    FI;
ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
  DEST ← SRC; (* Writes to selected I/O port *)
FI;

```

## Flags Affected

None.

## Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

## Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same as protected mode exceptions.

## 64-Bit Mode Exceptions

Same as protected mode exceptions.

## OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
6E	OUTS DX, <i>m8</i>	NP	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, <i>m16</i>	NP	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, <i>m32</i>	NP	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6E	OUTSB	NP	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSW	NP	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSD	NP	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

\*\* In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit mode, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.

### Instruction Operand Encoding

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

### Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:SI, DS:ESI or the RSI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI or RSI registers, which must be loaded correctly before the OUTS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the SI/ESI/RSI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the SI/ESI/RSI register is decremented.) The SI/ESI/RSI register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 16, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, the default operand size is 32 bits; operand size is not promoted by the use of REX.W. In 64-bit mode, the default address size is 64 bits, and 64-bit address is specified using RSI by default. 32-bit address using ESI is supported 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 *)
      F;
    ELSE (Real Mode or Protected Mode or 64-Bit Mode with CPL ≤ IOPL *)
      DEST ← SRC; (* Writes to I/O port *)
    F;

```

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;
          F;
        ELSE (* 32-Bit Address Size *)
          IF DF = 0
            THEN   ESI ← ESI + 1;
            ELSE   ESI ← ESI - 1;
          F;
        F;
      ELSE
        IF DF = 0
          THEN   (E)SI ← (E)SI + 1;
          ELSE   (E)SI ← (E)SI - 1;
        F;
      F;
  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;
        FI;
    ELSE
        IF DF = 0
            THEN (E)SI ← (E)SI + 4;
            ELSE (E)SI ← (E)SI - 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 a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS segment.  
If the segment register contains a NULL segment selector.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## PABSB/PABSW/PABSD — Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 1C /r <sup>1</sup> PABSB <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Compute the absolute value of bytes in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1C /r PABSB <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Compute the absolute value of bytes in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
0F 38 1D /r <sup>1</sup> PABSW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1D /r PABSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
0F 38 1E /r <sup>1</sup> PABSD <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Compute the absolute value of 32-bit integers in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1E /r PABSD <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Compute the absolute value of 32-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1C /r VPABSB <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Compute the absolute value of bytes in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1D /r VPABSW <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Compute the absolute value of 16-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1E /r VPABSD <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Compute the absolute value of 32-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.256.66.0F38.WIG 1C /r VPABSB <i>ymm1, ymm2/m256</i>	RM	V/V	AVX2	Compute the absolute value of bytes in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .
VEX.256.66.0F38.WIG 1D /r VPABSW <i>ymm1, ymm2/m256</i>	RM	V/V	AVX2	Compute the absolute value of 16-bit integers in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .
VEX.256.66.0F38.WIG 1E /r VPABSD <i>ymm1, ymm2/m256</i>	RM	V/V	AVX2	Compute the absolute value of 32-bit integers in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .

**NOTES:**

- See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

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

## Description

(V)PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). (V)PABSB operates on signed bytes, (V)PABSW operates on 16-bit words, and (V)PABSD operates on signed 32-bit integers. The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register, a YMM register, a 128-bit memory location, or a 256-bit memory location. The destination operand can be an MMX, an XMM or a YMM register. Both operands can be MMX registers or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16byte boundary or a general-protection exception (#GP) will be generated.

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

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

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

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

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

## Operation

### PABSB (with 64 bit operands)

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

### PABSB (with 128 bit operands)

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

### PABSW (with 64 bit operands)

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

### PABSW (with 128 bit operands)

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

### PABSD (with 64 bit operands)

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

### PABSD (with 128 bit operands)

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

### PABSB (128-bit Legacy SSE version)

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

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

DEST[127:0]  $\leftarrow$  BYTE\_ABS(SRC)  
 DEST[VLMAX-1:128]  $\leftarrow$  0

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

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

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

DEST[127:0]  $\leftarrow$  WORD\_ABS(SRC)  
 DEST[VLMAX-1:128] (Unmodified)

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

DEST[127:0]  $\leftarrow$  WORD\_ABS(SRC)  
 DEST[VLMAX-1:128]  $\leftarrow$  0

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

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

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

DEST[127:0]  $\leftarrow$  DWORD\_ABS(SRC)  
 DEST[VLMAX-1:128] (Unmodified)

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

DEST[127:0]  $\leftarrow$  DWORD\_ABS(SRC)  
 DEST[VLMAX-1:128]  $\leftarrow$  0

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

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

**Intel C/C++ Compiler Intrinsic Equivalents**

PABSB:	<code>__m64 _mm_abs_pi8 (__m64 a)</code>
(V)PABSB:	<code>__m128i _mm_abs_epi8 (__m128i a)</code>
VPABSB:	<code>__m256i _mm256_abs_epi8 (__m256i a)</code>
PABSW:	<code>__m64 _mm_abs_pi16 (__m64 a)</code>
(V)PABSW:	<code>__m128i _mm_abs_epi16 (__m128i a)</code>
VPABSW:	<code>__m256i _mm256_abs_epi16 (__m256i a)</code>
PABSD:	<code>__m64 _mm_abs_pi32 (__m64 a)</code>
(V)PABSD:	<code>__m128i _mm_abs_epi32 (__m128i a)</code>
VPABSD:	<code>__m256i _mm256_abs_epi32 (__m256i a)</code>

**SIMD Floating-Point Exceptions**

None.

### Other Exceptions

See Exceptions Type 4; additionally

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

## PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 63 / <sup>1</sup> PACKSSWB <i>mm1, mm2/m64</i>	RM	V/V	MMX	Converts 4 packed signed word integers from <i>mm1</i> and from <i>mm2/m64</i> into 8 packed signed byte integers in <i>mm1</i> using signed saturation.
66 OF 63 / <i>r</i> PACKSSWB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Converts 8 packed signed word integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
OF 6B / <sup>1</sup> PACKSSDW <i>mm1, mm2/m64</i>	RM	V/V	MMX	Converts 2 packed signed doubleword integers from <i>mm1</i> and from <i>mm2/m64</i> into 4 packed signed word integers in <i>mm1</i> using signed saturation.
66 OF 6B / <i>r</i> PACKSSDW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Converts 4 packed signed doubleword integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.NDS.128.66.0F.WIG 63 / <i>r</i> VPACKSSWB <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Converts 8 packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
VEX.NDS.128.66.0F.WIG 6B / <i>r</i> VPACKSSDW <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Converts 4 packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.NDS.256.66.0F.WIG 63 / <i>r</i> VPACKSSWB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Converts 16 packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 32 packed signed byte integers in <i>ymm1</i> using signed saturation.
VEX.NDS.256.66.0F.WIG 6B / <i>r</i> VPACKSSDW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Converts 8 packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 16 packed signed word integers in <i>ymm1</i> using signed saturation.

### NOTES:

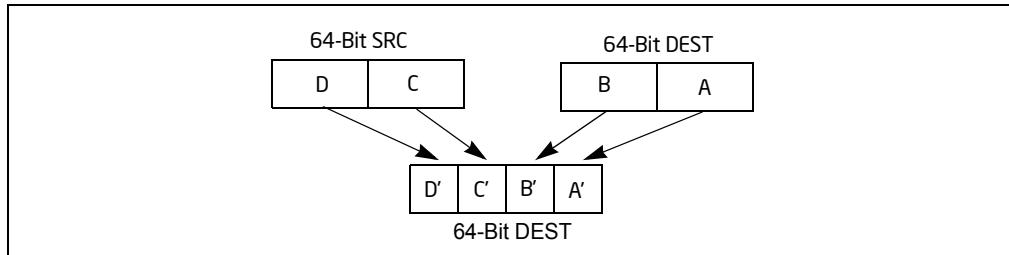
1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

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

### Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-2 for an example of the packing operation.



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

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

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

The (V)PACKSSWB and (V)PACKSSDW instructions operate on either 64-bit, 128-bit operands or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

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

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

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

## Operation

### PACKSSWB (with 64-bit operands)

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

### PACKSSDW (with 64-bit operands)

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

**PACKSSWB instruction (128-bit Legacy SSE version)**

```

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

```

**PACKSSDW instruction (128-bit Legacy SSE version)**

```

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

```

**VPACKSSWB instruction (VEX.128 encoded version)**

```

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

```

**VPACKSSDW instruction (VEX.128 encoded version)**

```

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);

```

```
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[VLMAX-1:128] ← 0;
```

**VPACKSSWB instruction (VEX.256 encoded version)**

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

**VPACKSSDW instruction (VEX.256 encoded version)**

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

DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);

### Intel C/C++ Compiler Intrinsic Equivalents

PACKSSWB:  $\text{__m64 } \text{mm\_packs\_pi16}(\text{__m64 m1}, \text{__m64 m2})$   
 (V)PACKSSWB:  $\text{__m128i } \text{mm\_packs\_epi16}(\text{__m128i m1}, \text{__m128i m2})$   
 VPACKSSWB:  $\text{__m256i } \text{mm256\_packs\_epi16}(\text{__m256i m1}, \text{__m256i m2})$   
 PACKSSDW:  $\text{__m64 } \text{mm\_packs\_pi32}(\text{__m64 m1}, \text{__m64 m2})$   
 (V)PACKSSDW:  $\text{__m128i } \text{mm\_packs\_epi32}(\text{__m128i m1}, \text{__m128i m2})$   
 VPACKSSDW:  $\text{__m256i } \text{mm256\_packs\_epi32}(\text{__m256i m1}, \text{__m256i m2})$

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

**PACKUSDW – Pack with Unsigned Saturation**

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

**Instruction Operand Encoding**

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

**Description**

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

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

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

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

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

**Operation****PACKUSDW (Legacy SSE instruction)**

```

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

```

```

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

```

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

```

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[63:48] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[VLMAX-1:128] ← 0;

```

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

```

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[63:48] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[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];

```

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

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

**Flags Affected**

None.

**SIMD Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 67 /r <sup>1</sup> PACKUSWB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Converts 4 signed word integers from <i>mm</i> and 4 signed word integers from <i>mm/m64</i> into 8 unsigned byte integers in <i>mm</i> using unsigned saturation.
66 OF 67 /r PACKUSWB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Converts 8 signed word integers from <i>xmm1</i> and 8 signed word integers from <i>xmm2/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Converts 8 signed word integers from <i>xmm2</i> and 8 signed word integers from <i>xmm3/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Converts 16 signed word integers from <i>ymm2</i> and 16 signed word integers from <i>ymm3/m256</i> into 32 unsigned byte integers in <i>ymm1</i> using unsigned saturation.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

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

### Description

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

The PACKUSWB instruction operates on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

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

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

## Operation

### PACKUSWB (with 64-bit operands)

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

### PACKUSWB (Legacy SSE instruction)

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

### PACKUSWB (VEX.128 encoded version)

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

### VPACKUSWB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
```

```

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

```

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

PACKUSWB:    \_\_m64 \_mm\_packs\_pu16(\_\_m64 m1, \_\_m64 m2)  
 (V)PACKUSWB: \_\_m128i \_mm\_packus\_epi16(\_\_m128i m1, \_\_m128i m2)  
 VPACKUSWB:    \_\_m256i \_mm256\_packus\_epi16(\_\_m256i m1, \_\_m256i m2);

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                      If VEX.L = 1.

**PADDB/PADDW/PADDD—Add Packed Integers**

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

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

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

**Description**

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

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

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

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

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

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

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

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

## Operation

### PADDB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] + SRC[63:56];
```

### PADDB (with 128-bit operands)

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

### VPADDB (VEX.128 encoded version)

```
DEST[7:0] ← SRC1[7:0]+SRC2[7:0]
DEST[15:8] ← SRC1[15:8]+SRC2[15:8]
DEST[23:16] ← SRC1[23:16]+SRC2[23:16]
DEST[31:24] ← SRC1[31:24]+SRC2[31:24]
DEST[39:32] ← SRC1[39:32]+SRC2[39:32]
DEST[47:40] ← SRC1[47:40]+SRC2[47:40]
DEST[55:48] ← SRC1[55:48]+SRC2[55:48]
DEST[63:56] ← SRC1[63:56]+SRC2[63:56]
DEST[71:64] ← SRC1[71:64]+SRC2[71:64]
DEST[79:72] ← SRC1[79:72]+SRC2[79:72]
DEST[87:80] ← SRC1[87:80]+SRC2[87:80]
DEST[95:88] ← SRC1[95:88]+SRC2[95:88]
DEST[103:96] ← SRC1[103:96]+SRC2[103:96]
DEST[111:104] ← SRC1[111:104]+SRC2[111:104]
DEST[119:112] ← SRC1[119:112]+SRC2[119:112]
DEST[127:120] ← SRC1[127:120]+SRC2[127:120]
DEST[VLMAX-1:128] ← 0
```

### VPADDB (VEX.256 encoded instruction)

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

**PADDW (with 64-bit operands)**

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd and 3rd word *)
DEST[63:48] ← DEST[63:48] + SRC[63:48];
```

**PADDW (with 128-bit operands)**

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

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

```
DEST[15:0] ← SRC1[15:0]+SRC2[15:0]
DEST[31:16] ← SRC1[31:16]+SRC2[31:16]
DEST[47:32] ← SRC1[47:32]+SRC2[47:32]
DEST[63:48] ← SRC1[63:48]+SRC2[63:48]
DEST[79:64] ← SRC1[79:64]+SRC2[79:64]
DEST[95:80] ← SRC1[95:80]+SRC2[95:80]
DEST[111:96] ← SRC1[111:96]+SRC2[111:96]
DEST[127:112] ← SRC1[127:112]+SRC2[127:112]
DEST[VLMAX-1:128] ← 0
```

**VPADDW (VEX.256 encoded instruction)**

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

**PADDD (with 64-bit operands)**

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

**PADDD (with 128-bit operands)**

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

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

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

**VPADDD (VEX.256 encoded instruction)**

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

**Intel C/C++ Compiler Intrinsic Equivalents**

PADDB:	<code>_m64 _mm_add_pi8(_m64 m1, _m64 m2)</code>
(V)PADDB:	<code>_m128i _mm_add_epi8 (_m128ia,_m128ib )</code>
VPADDB:	<code>_m256i _mm256_add_epi8 (_m256ia,_m256i b )</code>
PADDW:	<code>_m64 _mm_add_pi16(_m64 m1, _m64 m2)</code>
(V)PADDW:	<code>_m128i _mm_add_epi16 ( __m128i a, __m128i b)</code>

VPADDW:        $\text{__m256i mm256\_add\_epi16}(\text{__m256i } a, \text{__m256i } b)$   
PADD:        $\text{__m64 mm\_add\_pi32}(\text{__m64 } m1, \text{__m64 } m2)$   
(V)PADDD:     $\text{__m128i mm\_add\_epi32}(\text{__m128i } a, \text{__m128i } b)$   
VPADDD:       $\text{__m256i mm256\_add\_epi32}(\text{__m256i } a, \text{__m256i } b)$

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

## PADDQ—Add Packed Quadword Integers

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

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

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

### Description

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

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

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

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

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

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

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

### Operation

#### PADDQ (with 64-Bit operands)

DEST[63:0] ← DEST[63:0] + SRC[63:0];

**PADDQ (with 128-Bit operands)**

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

**VPADDQ (VEX.128 encoded instruction)**

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

**VPADDQ (VEX.256 encoded instruction)**

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

**Intel C/C++ Compiler Intrinsic Equivalents**

PADDQ:	<code>__m64 _mm_add_si64 (__m64 a, __m64 b)</code>
(V)PADDQ:	<code>__m128i _mm_add_epi64 (__m128i a, __m128i b)</code>
VPADDQ:	<code>__m256i _mm256_add_epi64 (__m256i a, __m256i b)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

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

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

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

### Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

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

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

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

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

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

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

## Operation

### PADDSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte(DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte(DEST[63:56] + SRC[63:56]);
```

### PADDSB (with 128-bit operands)

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

### VPADDSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

### VPADDSB (VEX.256 encoded version)

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

### PADDSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToSignedWord(DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord(DEST[63:48] + SRC[63:48]);
```

### PADDSW (with 128-bit operands)

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

### VPADDSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

### VPADDSW (VEX.256 encoded version)

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

**Intel C/C++ Compiler Intrinsic Equivalents**

PADDSB:   `__m64 _mm_adds_pi8(__m64 m1, __m64 m2)`  
 (V)PADDSB:   `__m128i _mm_adds_epi8 ( __m128i a, __m128i b)`  
 VPADDSB:   `__m256i _mm256_adds_epi8 ( __m256i a, __m256i b)`  
 PADDSW:   `__m64 _mm_adds_pi16(__m64 m1, __m64 m2)`  
 (V)PADDSW:   `__m128i _mm_adds_epi16 ( __m128i a, __m128i b)`  
 VPADDSW:   `__m256i _mm256_adds_epi16 ( __m256i a, __m256i b)`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                         If VEX.L = 1.

## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

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

### NOTES:

- See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

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

### Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

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

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

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX tech-

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

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

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

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

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

## Operation

### PADDUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte(DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte(DEST[63:56] + SRC[63:56])
```

### PADDUSB (with 128-bit operands)

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

### VPADDUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

### VPADDUSB (VEX.256 encoded version)

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

### PADDUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord(DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord(DEST[63:48] + SRC[63:48]);
```

### PADDUSW (with 128-bit operands)

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

### VPADDUSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

### VPADDUSW (VEX.256 encoded version)

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

**Intel C/C++ Compiler Intrinsic Equivalents**

PADDUSB:    \_\_m64 \_mm\_adds\_pu8(\_\_m64 m1, \_\_m64 m2)  
 PADDUSW:    \_\_m64 \_mm\_adds\_pu16(\_\_m64 m1, \_\_m64 m2)  
 (V)PADDUSB: \_\_m128i \_mm\_adds\_epu8 ( \_\_m128i a, \_\_m128i b )  
 (V)PADDUSW: \_\_m128i \_mm\_adds\_epu16 ( \_\_m128i a, \_\_m128i b )  
 VPADDUSB:   \_\_m256i \_mm256\_adds\_epu8 ( \_\_m256i a, \_\_m256i b )  
 VPADDUSW:   \_\_m256i \_mm256\_adds\_epu16 ( \_\_m256i a, \_\_m256i b )

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                      If VEX.L = 1.

**PALIGNR — Packed Align Right**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 3A OF /r ib <sup>1</sup> PALIGNR <i>mm1, mm2/m64, imm8</i>	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>mm1</i> .
66 OF 3A OF /r ib PALIGNR <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG OF /r ib VPALIGNR <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Concatenate <i>xmm2</i> and <i>xmm3/m128</i> , extract byte aligned result shifted to the right by constant value in <i>imm8</i> and result is stored in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG OF /r ib VPALIGNR <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Concatenate pairs of 16 bytes in <i>ymm2</i> and <i>ymm3/m256</i> into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in <i>imm8</i> from each intermediate result, and two 16-byte results are stored in <i>ymm1</i> .

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX, XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e. 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

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

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

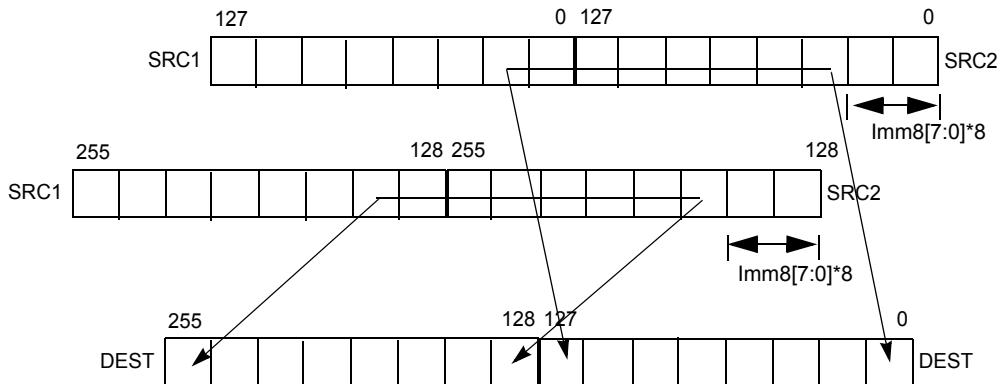
VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand.

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

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



**Figure 4-3. 256-bit VPALIGN Instruction Operation**

## Operation

### PALIGNR (with 64-bit operands)

```
temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[63:0] = temp1[63:0]
```

### PALIGNR (with 128-bit operands)

```
temp1[255:0] <- ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] <- temp1[127:0]
DEST[VLMAX-1:128] (Unmodified)
```

### VPALIGNR (VEX.128 encoded version)

```
temp1[255:0] <- ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] <- temp1[127:0]
DEST[VLMAX-1:128] <- 0
```

### VPALIGNR (VEX.256 encoded version)

```
temp1[255:0] <- ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] <- temp1[127:0]
temp1[255:0] <- ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[255:128] <- temp1[127:0]
```

## Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR:	<code>__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)</code>
(V)PALIGNR:	<code>__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)</code>
VPALIGNR:	<code>__m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)</code>

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PAND—Logical AND

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

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

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

### Description

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

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

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PAND (128-bit Legacy SSE version)

DEST  $\leftarrow$  DEST AND SRC

DEST[VLMAX-1:128] (Unmodified)

#### VPAND (VEX.128 encoded version)

DEST  $\leftarrow$  SRC1 AND SRC2

DEST[VLMAX-1:128]  $\leftarrow$  0

**VPAND (VEX.256 encoded instruction)**

$\text{DEST}[255:0] \leftarrow (\text{SRC1}[255:0] \text{ AND } \text{SRC2}[255:0])$

**Intel C/C++ Compiler Intrinsic Equivalent**

PAND:	<code>__m64 _mm_and_si64 (__m64 m1, __m64 m2)</code>
(V)PAND:	<code>__m128i _mm_and_si128 ( __m128i a, __m128i b)</code>
VPAND:	<code>__m256i _mm256_and_si256 ( __m256i a, __m256i b)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF DF /r <sup>1</sup>	RM	V/V	MMX	Bitwise AND NOT of <i>mm/m64</i> and <i>mm</i> .
PANDN <i>mm, mm/m64</i>				
66 OF DF /r	RM	V/V	SSE2	Bitwise AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .
PANDN <i>xmm1, xmm2/m128</i>				
VEX.NDS.128.66.0F.WIG DF /r	RVM	V/V	AVX	Bitwise AND NOT of <i>xmm3/m128</i> and <i>xmm2</i> .
VPANDN <i>xmm1, xmm2, xmm3/m128</i>				
VEX.NDS.256.66.0F.WIG DF /r	RVM	V/V	AVX2	Bitwise AND NOT of <i>ymm2</i> , and <i>ymm3/m256</i> and store result in <i>ymm1</i> .
VPANDN <i>ymm1, ymm2, ymm3/m256</i>				

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PANDN(128-bit Legacy SSE version)

DEST ← NOT(DEST) AND SRC

DEST[VLMAX-1:128] (Unmodified)

#### VPANDN (VEX.128 encoded version)

DEST ← NOT(SRC1) AND SRC2

DEST[VLMAX-1:128] ← 0

**VPANDN (VEX.256 encoded instruction)**

$\text{DEST}[255:0] \leftarrow ((\text{NOT } \text{SRC1}[255:0]) \text{ AND } \text{SRC2}[255:0])$

**Intel C/C++ Compiler Intrinsic Equivalent**

PANDN:	<code>__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)</code>
(V)PANDN:	<code>__m128i _mm_andnot_si128 ( __m128i a, __m128i b)</code>
VPANDN:	<code>__m256i _mm256_andnot_si256 ( __m256i a, __m256i b)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PAUSE—Spin Loop Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 90	PAUSE	NP	Valid	Valid	Gives hint to processor that improves performance of spin-wait loops.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Improves the performance of spin-wait loops. When executing a “spin-wait loop,” processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor’s power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

Execute\_Next\_Instruction(Delay);

### Numeric Exceptions

None.

### Exceptions (All Operating Modes)

#UD                    If the LOCK prefix is used.

**PAVGB/PAVGW—Average Packed Integers**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 bit Mode Support</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
OF EO /r <sup>1</sup> PAVGB mm1, mm2/m64	RM	V/V	SSE	Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.
66 OF EO, /r PAVGW xmm1, xmm2/m128	RM	V/V	SSE2	Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding.
OF E3 /r <sup>1</sup> PAVGW mm1, mm2/m64	RM	V/V	SSE	Average packed unsigned word integers from mm2/m64 and mm1 with rounding.
66 OF E3 /r PAVGW xmm1, xmm2/m128	RM	V/V	SSE2	Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding.
VEX.NDS.128.66.0F.WIG EO /r VPAVGB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Average packed unsigned byte integers from xmm3/m128 and xmm2 with rounding.
VEX.NDS.128.66.0F.WIG E3 /r VPAVGW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Average packed unsigned word integers from xmm3/m128 and xmm2 with rounding.
VEX.NDS.256.66.0F.WIG EO /r VPAVGB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1.
VEX.NDS.256.66.0F.WIG E3 /r VPAVGW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1.

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

## Operation

### PAVGB (with 64-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 6 *)
DEST[63:56] ← (SRC[63:56] + DEST[63:56] + 1) >> 1;
```

### PAVGW (with 64-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 and 3 *)
DEST[63:48] ← (SRC[63:48] + DEST[63:48] + 1) >> 1;
```

### PAVGB (with 128-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 14 *)
DEST[127:120] ← (SRC[127:120] + DEST[127:120] + 1) >> 1;
```

### PAVGW (with 128-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 6 *)
DEST[127:112] ← (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

### VPAVGB (VEX.128 encoded version)

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
DEST[VLMAX-1:128] ← 0
```

### VPAVGW (VEX.128 encoded version)

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[127:112] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1
DEST[VLMAX-1:128] ← 0
```

### VPAVGB (VEX.256 encoded instruction)

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 31)
DEST[255:248] ← (SRC1[255:248] + SRC2[255:248] + 1) >> 1;
```

### VPAVGW (VEX.256 encoded instruction)

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 15)
DEST[255:14] ← (SRC1[255:240] + SRC2[255:240] + 1) >> 1;
```

## Intel C/C++ Compiler Intrinsic Equivalent

PAVGB:	<code>__m64 _mm_avg_pu8 (__m64 a, __m64 b)</code>
PAVGW:	<code>__m64 _mm_avg_pu16 (__m64 a, __m64 b)</code>
(V)PAVGB:	<code>__m128i _mm_avg_epu8 ( __m128i a, __m128i b)</code>

(V)PAVGW:  $\text{__m128i}_\text{mm\_avg\_epu16}(\text{__m128i } a, \text{__m128i } b)$   
VPAVGB:  $\text{__m256i}_\text{mm256\_avg\_epu8}(\text{__m256i } a, \text{__m256i } b)$   
VPAVGW:  $\text{__m256i}_\text{mm256\_avg\_epu16}(\text{__m256i } a, \text{__m256i } b)$

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

## PBLENDVB – Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB <i>xmm1, xmm2/m128, &lt;XMM0&gt;</i>	RM	V/V	SSE4_1	Select byte values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in the high bit of each byte in <i>XMM0</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4C /r /is4 VPBLENDVB <i>xmm1, xmm2, xmm3/m128, xmm4</i>	RVMR	V/V	AVX	Select byte values from <i>xmm2</i> and <i>xmm3/m128</i> using mask bits in the specified mask register, <i>xmm4</i> , and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.W0 4C /r /is4 VPBLENDVB <i>ymm1, ymm2, ymm3/m256, ymm4</i>	RVMR	V/V	AVX2	Select byte values from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in the high bit of each byte in <i>ymm4</i> and store the values into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	<XMM0>	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

### Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, XMM0. The mask bits are the most significant bit in each byte element of the XMM0 register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

The register assignment of the implicit third operand is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (VLMAX-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source register is an YMM register and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

### Operation

#### PBLENDVB (128-bit Legacy SSE version)

```

MASK ← XMM0
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC[7:0];
ELSE DEST[7:0] ← DEST[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC[15:8];

```

```

ELSE DEST[15:8] ← DEST[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC[23:16]
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ← DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ← DEST[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120]);
DEST[VLMAX-1:128] (Unmodified)

```

**VPBLENDVB (VEX.128 encoded version)**

```

MASK ← SRC3
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC2[87:80]

```

```

ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120]);
DEST[VLMAX-1:128] ← 0

```

**VPBLENDVB (VEX.256 encoded version)**

```

MASK ← SRC3
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120]);
IF (MASK[135] == 1) THEN DEST[135:128] ← SRC2[135:128];
ELSE DEST[135:128] ← SRC1[135:128];
IF (MASK[143] == 1) THEN DEST[143:136] ← SRC2[143:136];
ELSE DEST[143:136] ← SRC1[143:136];
IF (MASK[151] == 1) THEN DEST[151:144] ← SRC2[151:144]
ELSE DEST[151:144] ← SRC1[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] ← SRC2[159:152]

```

```

ELSE DEST[159:152] ← SRC1[159:152];
IF (MASK[167] == 1) THEN DEST[167:160] ← SRC2[167:160]
ELSE DEST[167:160] ← SRC1[167:160];
IF (MASK[175] == 1) THEN DEST[175:168] ← SRC2[175:168]
ELSE DEST[175:168] ← SRC1[175:168];
IF (MASK[183] == 1) THEN DEST[183:176] ← SRC2[183:176]
ELSE DEST[183:176] ← SRC1[183:176];
IF (MASK[191] == 1) THEN DEST[191:184] ← SRC2[191:184]
ELSE DEST[191:184] ← SRC1[191:184];
IF (MASK[199] == 1) THEN DEST[199:192] ← SRC2[199:192]
ELSE DEST[199:192] ← SRC1[199:192];
IF (MASK[207] == 1) THEN DEST[207:200] ← SRC2[207:200]
ELSE DEST[207:200] ← SRC1[207:200]
IF (MASK[215] == 1) THEN DEST[215:208] ← SRC2[215:208]
ELSE DEST[215:208] ← SRC1[215:208];
IF (MASK[223] == 1) THEN DEST[223:216] ← SRC2[223:216]
ELSE DEST[223:216] ← SRC1[223:216];
IF (MASK[231] == 1) THEN DEST[231:224] ← SRC2[231:224]
ELSE DEST[231:224] ← SRC1[231:224];
IF (MASK[239] == 1) THEN DEST[239:232] ← SRC2[239:232]
ELSE DEST[239:232] ← SRC1[239:232];
IF (MASK[247] == 1) THEN DEST[247:240] ← SRC2[247:240]
ELSE DEST[247:240] ← SRC1[247:240];
IF (MASK[255] == 1) THEN DEST[255:248] ← SRC2[255:248]
ELSE DEST[255:248] ← SRC1[255:248]

```

### Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDVB:   `__m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);`  
 VPBLENDVB:   `__m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);`

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.W = 1.

## PBLENDW – Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A OE /r ib PBLENDW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select words from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG OE /r ib VPBLENDW <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Select words from <i>xmm2</i> and <i>xmm3/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG OE /r ib VPBLENDW <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Select words from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in <i>imm8</i> and store the values into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word element in the destination operand is unchanged.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### PBLENDW (128-bit Legacy SSE version)

```

IF (imm8[0] = 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC[31:16]
ELSE DEST[31:16] ← DEST[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC[47:32]
ELSE DEST[47:32] ← DEST[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC[63:48]
ELSE DEST[63:48] ← DEST[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC[79:64]
ELSE DEST[79:64] ← DEST[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC[95:80]
ELSE DEST[95:80] ← DEST[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC[111:96]
ELSE DEST[111:96] ← DEST[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC[127:112]
```

```
ELSE DEST[127:112] ← DEST[127:112]
```

**VPBLENDW (VEX.128 encoded version)**

```
IF (imm8[0] = 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
DEST[VLMAX-1:128] ← 0
```

**VPBLENDW (VEX.256 encoded version)**

```
IF (imm8[0] == 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
IF (imm8[0] == 1) THEN DEST[143:128] ← SRC2[143:128]
ELSE DEST[143:128] ← SRC1[143:128]
IF (imm8[1] == 1) THEN DEST[159:144] ← SRC2[159:144]
ELSE DEST[159:144] ← SRC1[159:144]
IF (imm8[2] == 1) THEN DEST[175:160] ← SRC2[175:160]
ELSE DEST[175:160] ← SRC1[175:160]
IF (imm8[3] == 1) THEN DEST[191:176] ← SRC2[191:176]
ELSE DEST[191:176] ← SRC1[191:176]
IF (imm8[4] == 1) THEN DEST[207:192] ← SRC2[207:192]
ELSE DEST[207:192] ← SRC1[207:192]
IF (imm8[5] == 1) THEN DEST[223:208] ← SRC2[223:208]
ELSE DEST[223:208] ← SRC1[223:208]
IF (imm8[6] == 1) THEN DEST[239:224] ← SRC2[239:224]
ELSE DEST[239:224] ← SRC1[239:224]
IF (imm8[7] == 1) THEN DEST[255:240] ← SRC2[255:240]
ELSE DEST[255:240] ← SRC1[255:240]
```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PBLENDW: `__m128i_mm_blend_ep16 (__m128i v1, __m128i v2, const int mask);`  
VPBLENDW: `__m256i_mm256_blend_ep16 (__m256i v1, __m256i v2, const int mask)`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

**PCLMULQDQ - Carry-Less Multiplication Quadword**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	CLMUL	Carry-less multiplication of one quadword of <i>xmm1</i> by one quadword of <i>xmm2/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm1</i> and <i>xmm2/m128</i> should be used.
VEX.NDS.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	Both CLMUL and AVX flags	Carry-less multiplication of one quadword of <i>xmm2</i> by one quadword of <i>xmm3/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm2</i> and <i>xmm3/m128</i> should be used.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-10, other bits of the immediate byte are ignored.

**Table 4-10. PCLMULQDQ Quadword Selection of Immediate Byte**

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL( SRC2 <sup>1</sup> [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:**

- SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for Imm8.

**Table 4-11. Pseudo-Op and PCLMULQDQ Implementation**

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ <i>xmm1, xmm2</i>	0000_0000B
PCLMULHQLQDQ <i>xmm1, xmm2</i>	0000_0001B
PCLMULLQHDQ <i>xmm1, xmm2</i>	0001_0000B
PCLMULHQHDQ <i>xmm1, xmm2</i>	0001_0001B

**Operation****PCLMULQDQ**

```

IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [i] ← (TEMP1[0] and TEMP2[i]);
    For j = 1 to i {
        TmpB [i] ← TmpB [i] xor (TEMP1[j] and TEMP2[i - j])
    }
    DEST[i] ← TmpB[i];
}
For i = 64 to 126 {
    TmpB [i] ← 0;
    For j = i - 63 to 63 {
        TmpB [i] ← TmpB [i] xor (TEMP1[j] and TEMP2[i - j])
    }
    DEST[i] ← TmpB[i];
}
DEST[127] ← 0;
DEST[VLMAX-1:128] (Unmodified)

```

**VPCLMULQDQ**

```

IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [i] ← (TEMP1[0] and TEMP2[i]);
    For j = 1 to i {
        TmpB [i] ← TmpB [i] xor (TEMP1[j] and TEMP2[i - j])
    }
    DEST[i] ← TmpB[i];
}
For i = 64 to 126 {
    TmpB [i] ← 0;
    For j = i - 63 to 63 {

```

```
    TmpB[i] ← TmpB[i] xor (TEMP1[j] and TEMP2[i - j])  
}  
DEST[i] ← TmpB[i];  
}  
DEST[VLMAX-1:127] ← 0;
```

### Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ:     \_\_m128i \_mm\_clmulepi64\_si128 (\_\_m128i, \_\_m128i, const int)

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4.

## PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 74 /r <sup>1</sup> PCMPEQB <i>mm, mm/m64</i>	RM	V/V	MMX	Compare packed bytes in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 74 /r PCMPEQB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Compare packed bytes in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
OF 75 /r <sup>1</sup> PCMPEQW <i>mm, mm/m64</i>	RM	V/V	MMX	Compare packed words in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 75 /r PCMPEQW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Compare packed words in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
OF 76 /r <sup>1</sup> PCMPEQD <i>mm, mm/m64</i>	RM	V/V	MMX	Compare packed doublewords in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 76 /r PCMPEQD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Compare packed doublewords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed bytes in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed words in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed doublewords in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW <i>ymm1, ymm2, ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed words in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD <i>ymm1, ymm2, ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed doublewords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ <i>ymm1, ymm2, ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed quadwords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE instructions:** The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

**128-bit Legacy SSE version:** The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded version:** The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

## Operation

### PCMPEQB (with 64-bit operands)

```

IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; Fl;
(* 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; Fl;
```

### PCMPEQB (with 128-bit operands)

```

IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; Fl;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] = SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; Fl;
```

### VPCMPEQB (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX-1:128] ← 0
```

### VPCMPEQB (VEX.256 encoded version)

```

DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
```

### PCMPEQW (with 64-bit operands)

```

IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; Fl;
(* 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; Fl;
```

### PCMPEQW (with 128-bit operands)

```

IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; Fl;
```

(\* Continue comparison of 2nd through 7th words in DEST and SRC \*)  
 IF DEST[127:112] = SRC[127:112]  
   THEN DEST[127:112] ← FFFFH;  
   ELSE DEST[127:112] ← 0; FI;

**VPCMPEQW (VEX.128 encoded version)**

DEST[127:0] ← COMPARE\_WORDS\_EQUAL(SRC1[127:0],SRC2[127:0])  
 DEST[VLMAX-1:128] ← 0

**VPCMPEQW (VEX.256 encoded version)**

DEST[127:0] ← COMPARE\_WORDS\_EQUAL(SRC1[127:0],SRC2[127:0])  
 DEST[255:128] ← COMPARE\_WORDS\_EQUAL(SRC1[255:128],SRC2[255:128])

**PCMPEQD (with 64-bit operands)**

IF DEST[31:0] = SRC[31:0]  
   THEN DEST[31:0] ← FFFFFFFFH;  
   ELSE DEST[31:0] ← 0; FI;  
 IF DEST[63:32] = SRC[63:32]  
   THEN DEST[63:32] ← FFFFFFFFH;  
   ELSE DEST[63:32] ← 0; FI;

**PCMPEQD (with 128-bit operands)**

IF DEST[31:0] = SRC[31:0]  
   THEN DEST[31:0] ← FFFFFFFFH;  
   ELSE DEST[31:0] ← 0; FI;  
 (\* Continue comparison of 2nd and 3rd doublewords in DEST and SRC \*)  
 IF DEST[127:96] = SRC[127:96]  
   THEN DEST[127:96] ← FFFFFFFFH;  
   ELSE DEST[127:96] ← 0; FI;

**VPCMPEQD (VEX.128 encoded version)**

DEST[127:0] ← COMPARE\_DWORDS\_EQUAL(SRC1[127:0],SRC2[127:0])  
 DEST[VLMAX-1:128] ← 0

**VPCMPEQD (VEX.256 encoded version)**

DEST[127:0] ← COMPARE\_DWORDS\_EQUAL(SRC1[127:0],SRC2[127:0])  
 DEST[255:128] ← COMPARE\_DWORDS\_EQUAL(SRC1[255:128],SRC2[255:128])

**Intel C/C++ Compiler Intrinsic Equivalents**

PCMPEQB:	<code>__m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)</code>
PCMPEQW:	<code>__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)</code>
PCMPEQD:	<code>__m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)</code>
(V)PCMPEQB:	<code>__m128i _mm_cmpeq_epi8 ( __m128i a, __m128i b)</code>
(V)PCMPEQW:	<code>__m128i _mm_cmpeq_epi16 ( __m128i a, __m128i b)</code>
(V)PCMPEQD:	<code>__m128i _mm_cmpeq_epi32 ( __m128i a, __m128i b)</code>
VPCMPEQB:	<code>__m256i _mm256_cmpeq_epi8 ( __m256i a, __m256i b)</code>
VPCMPEQW:	<code>__m256i _mm256_cmpeq_epi16 ( __m256i a, __m256i b)</code>
VPCMPEQD:	<code>__m256i _mm256_cmpeq_epi32 ( __m256i a, __m256i b)</code>

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PCMPEQQ – Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed qwords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed quadwords in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ <i>ymm1, ymm2, ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed quadwords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[63:0] = SRC[63:0])
  THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[63:0] ← 0; Fl;
IF (DEST[127:64] = SRC[127:64])
  THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[127:64] ← 0; Fl;

```

### VPCMPEQQ (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0

```

### VPCMPEQQ (VEX.256 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_EQUAL(SRC1[255:128],SRC2[255:128])

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

(V)PCMPEQQ: __m128i_mm_cmpeq_epi64(__m128i a, __m128i b);
VPCMPEQQ: __m256i_mm256_cmpeq_epi64(__m256i a, __m256i b);

```

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

## PCMPESTRI – Packed Compare Explicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 61 /r imm8 PCMPESTRI <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A.WIG 61 /r ib VPCMPESTRI <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

### Description

The instruction compares and processes data from two string fragments based on the encoded value in the Imm8 Control Byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates an index stored to the count register (ECX/RCX).

Each string fragment is represented by two values. The first value is an *xmm* (or possibly *m128* for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is *EAX/RAX* (for *xmm1*) or *EDX/RDX* (for *xmm2/m128*). The length represents the number of bytes/words which are valid for the respective *xmm/m128* data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of *imm8[bit3]* when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to *imm8[6]*) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
- SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
- OFlag – IntRes2[0]
- AFlag – Reset
- PFlag – Reset

### Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Length 1	Length 2	Result
16 bit	xmm	xmm/m128	EAX	EDX	ECX
32 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	RCX

### Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

```
int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);
```

**Intel C/C++ Compiler Intrinsiccs For Reading EFlag Results**

```
int _mm_cmpestra (_m128i a, int la, _m128i b, int lb, const int mode);
int _mm_cmpestrc (_m128i a, int la, _m128i b, int lb, const int mode);
int _mm_cmpestro (_m128i a, int la, _m128i b, int lb, const int mode);
int _mm_cmpestrs (_m128i a, int la, _m128i b, int lb, const int mode);
int _mm_cmpestrz (_m128i a, int la, _m128i b, int lb, const int mode);
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD                  If VEX.L = 1.  
                      If VEX.vvvv != 1111B.

## PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 60 /r imm8 PCMPESTRM <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in <i>XMM0</i>
VEX.128.66.0F3A.WIG 60 /r ib VPCMPESTRM <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in <i>XMM0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

### Description

The instruction compares data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates a mask stored to *XMM0*.

Each string fragment is represented by two values. The first value is an *xmm* (or possibly *m128* for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is *EAX/RAX* (for *xmm1*) or *EDX/RDX* (for *xmm2/m128*). The length represents the number of bytes/words which are valid for the respective *xmm/m128* data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of *imm8[bit3]* when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by *imm8[6]*, *IntRes2* is then either stored to the least significant bits of *XMM0* (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to *XMM0*.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if *IntRes2* is equal to zero, set otherwise
- ZFlag – Set if absolute-value of *EDX* is < 16 (8), reset otherwise
- SFlag – Set if absolute-value of *EAX* is < 16 (8), reset otherwise
- OFlag – *IntRes2[0]*
- AFlag – Reset
- PFlag – Reset

Note: In VEX.128 encoded versions, bits (VLMAX-1:128) of *XMM0* are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

## Effective Operand Size

Operating mode/size	Operand1	Operand 2	Length1	Length2	Result
16 bit	xmm	xmm/m128	EAX	EDX	XMM0
32 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	XMM0

## Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

```
__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);
```

## Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```
int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);
```

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD                    If VEX.L = 1.  
                      If VEX.vvvv != 1111B.

## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 64 /r <sup>1</sup> PCMPGTB <i>mm, mm/m64</i>	RM	V/V	MMX	Compare packed signed byte integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 64 /r PCMPGTB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
OF 65 /r <sup>1</sup> PCMPGTW <i>mm, mm/m64</i>	RM	V/V	MMX	Compare packed signed word integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 65 /r PCMPGTW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
OF 66 /r <sup>1</sup> PCMPGTD <i>mm, mm/m64</i>	RM	V/V	MMX	Compare packed signed doubleword integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 66 /r PCMPGTD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed doubleword integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMMPGTB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMMPGTW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMMPGTD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed doubleword integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMMPGTB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMMPGTW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
VEX.NDS.256.66.0F.WIG 66 /r VPCMMPGTD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed doubleword integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding data element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source

operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE instructions:** The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

**128-bit Legacy SSE version:** The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded version:** The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

## Operation

### PCMPGTB (with 64-bit operands)

```

IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; Fl;
(* 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; Fl;
```

### PCMPGTB (with 128-bit operands)

```

IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; Fl;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] > SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; Fl;
```

### VPCMPGTB (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_BYT_ES_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

### VPCMPGTB (VEX.256 encoded version)

```

DEST[127:0] ← COMPARE_BYT_ES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_BYT_ES_GREATER(SRC1[255:128],SRC2[255:128])
```

### PCMPGTW (with 64-bit operands)

```

IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; Fl;
(* 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; Fl;
```

**PCMPGTW (with 128-bit operands)**

```

IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; Fl;
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[63:48] > SRC[127:112]
    THEN DEST[127:112] ← FFFFH;
    ELSE DEST[127:112] ← 0; Fl;

```

**VPCMPGTW (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0

```

**VPCMPGTW (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])

```

**PCMPGTD (with 64-bit operands)**

```

IF DEST[31:0] > SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; Fl;
IF DEST[63:32] > SRC[63:32]
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; Fl;

```

**PCMPGTD (with 128-bit operands)**

```

IF DEST[31:0] > SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; Fl;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
IF DEST[127:96] > SRC[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; Fl;

```

**VPCMPGTD (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0

```

**VPCMPGTD (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])

```

**Intel C/C++ Compiler Intrinsic Equivalents**

PCMPGTB:	<code>_m64 _mm_cmpgt_pi8 (_m64 m1, _m64 m2)</code>
PCMPGTW:	<code>_m64 _mm_pcmpgt_pi16 (_m64 m1, _m64 m2)</code>
DCMPGTD:	<code>_m64 _mm_pcmpgt_pi32 (_m64 m1, _m64 m2)</code>
(V)PCMPGTB:	<code>_m128i _mm_cmpgt_epi8 (_m128i a, _m128i b)</code>
(V)PCMPGTW:	<code>_m128i _mm_cmpgt_epi16 (_m128i a, _m128i b)</code>
(V)DCMPGTD:	<code>_m128i _mm_cmpgt_epi32 (_m128i a, _m128i b)</code>
VPCMPGTB:	<code>_m256i _mm256_cmpgt_epi8 (_m256i a, _m256i b)</code>
VPCMPGTW:	<code>_m256i _mm256_cmpgt_epi16 (_m256i a, _m256i b)</code>
VPCMPGTD:	<code>_m256i _mm256_cmpgt_epi32 (_m256i a, _m256i b)</code>

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PCMPGTQ — Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ <i>xmm1,xmm2/m128</i>	RM	V/V	SSE4_2	Compare packed signed qwords in <i>xmm2/m128</i> and <i>xmm1</i> for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed qwords in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.256.66.0F38.WIG 37 /r VPCMPGTQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed qwords in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[63-0] > SRC[63-0])
  THEN DEST[63-0] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[63-0] ← 0; FI
IF (DEST[127-64] > SRC[127-64])
  THEN DEST[127-64] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[127-64] ← 0; FI

```

### VPCMPGTQ (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0

```

### VPCMPGTQ (VEX.256 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_GREATER(SRC1[255:128],SRC2[255:128])

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PCMPGTQ:    \_\_m128i \_mm\_cmplt\_epi64(\_\_m128i a, \_\_m128i b)  
VPCMGTQ:    \_\_m256i \_mm256\_cmplt\_epi64( \_\_m256i a, \_\_m256i b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD               If VEX.L = 1.

## PCMPISTRI — Packed Compare Implicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 63 /r imm8 PCMPISTRI <i>xmm1, xmm2/m128, imm8</i>	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 <i>xmm1, xmm2/m128, imm8</i>	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

### Description

The instruction compares data from two strings based on the encoded value in the Imm8 Control Byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates an index stored to ECX.

Each string is represented by a single value. The value is an *xmm* (or possibly *m128* for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if any byte/word of *xmm2/mem128* is null, reset otherwise
- SFlag – Set if any byte/word of *xmm1* is null, reset otherwise
- OFlag – IntRes2[0]
- AFlag – Reset
- PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	ECX
32 bit	xmm	xmm/m128	ECX
64 bit	xmm	xmm/m128	ECX
64 bit + REX.W	xmm	xmm/m128	RCX

**Intel C/C++ Compiler Intrinsic Equivalent For Returning Index**

```
int _mm_cmplibtri (_m128i a, _m128i b, const int mode);
```

**Intel C/C++ Compiler Intrinsics For Reading EFlag Results**

```
int _mm_cmplibstra (_m128i a, _m128i b, const int mode);
int _mm_cmplibstrc (_m128i a, _m128i b, const int mode);
int _mm_cmplibstro (_m128i a, _m128i b, const int mode);
int _mm_cmplibstrs (_m128i a, _m128i b, const int mode);
int _mm_cmplibstrz (_m128i a, _m128i b, const int mode);
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD                    If VEX.L = 1.  
                      If VEX.vvvv != 1111B.

## PCMPISTRM – Packed Compare Implicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 62 /r imm8 PCMPISTRM <i>xmm1, xmm2/m128, imm8</i>	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in <i>XMM0</i> .
VEX.128.66.0F3A.WIG 62 /r ib VPCMPISTRM <i>xmm1, xmm2/m128, imm8</i>	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in <i>XMM0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

### Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”) generating a mask stored to XMM0.

Each string is represented by a single value. The value is an *xmm* (or possibly *m128* for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operation are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if any byte/word of *xmm2/mem128* is null, reset otherwise
- SFlag – Set if any byte/word of *xmm1* is null, reset otherwise
- OFlag – IntRes2[0]
- AFlag – Reset
- PFlag – Reset

Note: In VEX.128 encoded versions, bits (VLMAX-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	XMM0
32 bit	xmm	xmm/m128	XMM0
64 bit	xmm	xmm/m128	XMM0
64 bit + REX.W	xmm	xmm/m128	XMM0

### Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

```
_m128i _mm_cmplstrm (_m128i a, _m128i b, const int mode);
```

**Intel C/C++ Compiler Intrinsiccs For Reading EFlag Results**

```
int _mm_cmplistra (_m128i a, _m128i b, const int mode);
int _mm_cmplistrc (_m128i a, _m128i b, const int mode);
int _mm_cmplistro (_m128i a, _m128i b, const int mode);
int _mm_cmplistrs (_m128i a, _m128i b, const int mode);
int _mm_cmplistrz (_m128i a, _m128i b, const int mode);
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD                  If VEX.L = 1.  
                      If VEX.vvvv != 1111B.

## PDEP – Parallel Bits Deposit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F2.0F38.W0 F5 /r PDEP r32a, r32b, r/m32	RVM	V/V	BMI2	Parallel deposit of bits from r32b using mask in r/m32, result is written to r32a.
VEX.NDS.LZ.F2.0F38.W1 F5 /r PDEP r64a, r64b, r/m64	RVM	V/N.E.	BMI2	Parallel deposit of bits from r64b using mask in r/m64, result is written to r64a.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

PDEP uses a mask in the second source operand (the third operand) to transfer/scatter contiguous low order bits in the first source operand (the second operand) into the destination (the first operand). PDEP takes the low bits from the first source operand and deposit them in the destination operand at the corresponding bit locations that are set in the second source operand (mask). All other bits (bits not set in mask) in destination are set to zero.

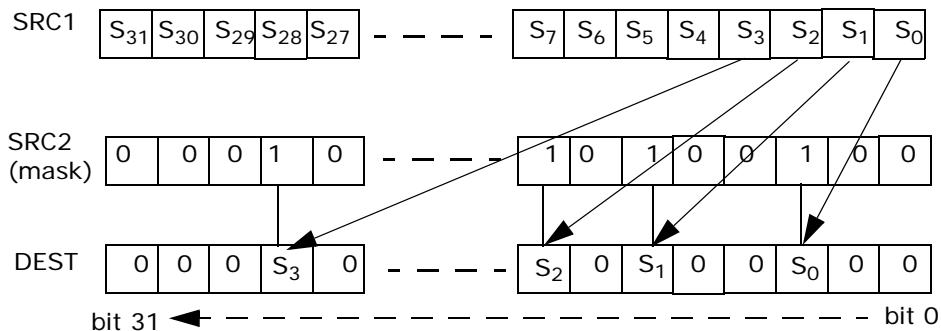


Figure 4-4. PDEP Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```

TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0 ;
m← 0, k← 0;
DO WHILE m< OperandSize

```

```

  IF MASK[ m ] = 1 THEN
    DEST[ m ] ← TEMP[ k ];
    k ← k+ 1;
  FI
  m ← m+ 1;

```

OD

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

PDEP:        `unsigned __int32 _pdep_u32(unsigned __int32 src, unsigned __int32 mask);`  
PDEP:        `unsigned __int64 _pdep_u64(unsigned __int64 src, unsigned __int32 mask);`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
#UD              If VEX.W = 1.

## PEXT – Parallel Bits Extract

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F3.0F38.W0 F5 /r PEXT r32a, r32b, r/m32	RVM	V/V	BMI2	Parallel extract of bits from r32b using mask in r/m32, result is written to r32a.
VEX.NDS.LZ.F3.0F38.W1 F5 /r PEXT r64a, r64b, r/m64	RVM	V/N.E.	BMI2	Parallel extract of bits from r64b using mask in r/m64, result is written to r64a.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

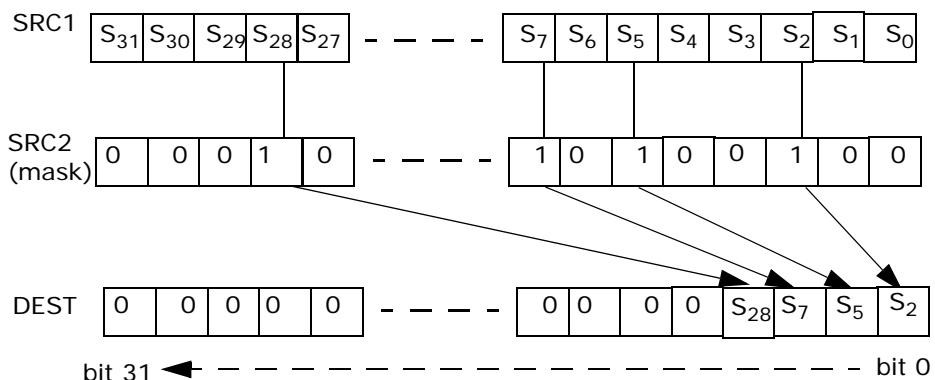


Figure 4-5. PEXT Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0 ;
m← 0, k← 0;
DO WHILE m< OperandSize
```

```
IF MASK[ m] = 1 THEN
  DEST[ k] ← TEMP[ m];
  k ← k+ 1;
```

Fl  
m ← m+ 1;

OD

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

PEXT:      unsigned \_\_int32 \_pext\_u32(unsigned \_\_int32 src, unsigned \_\_int32 mask);  
PEXT:      unsigned \_\_int64 \_pext\_u64(unsigned \_\_int64 src, unsigned \_\_int32 mask);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD              If VEX.W = 1.

**PEXTRB/PEXTRD/PEXTRQ – Extract Byte/Dword/Qword**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 bit Mode Support</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
66 0F 3A 14 /r ib PEXTRB <i>reg/m8, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of r32 or r64 are zeroed.
66 0F 3A 16 /r ib PEXTRD <i>r/m32, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r/m32</i> .
66 REX.W 0F 3A 16 /r ib PEXTRQ <i>r/m64, xmm2, imm8</i>	MRI	V/N.E.	SSE4_1	Extract a qword integer value from <i>xmm2</i> at the source qword offset specified by <i>imm8</i> into <i>r/m64</i> .
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB <i>reg/m8, xmm2, imm8</i>	MRI	V <sup>1</sup> /V	AVX	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD <i>r32/m32, xmm2, imm8</i>	MRI	V/V	AVX	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r32/m32</i> .
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ <i>r64/m64, xmm2, imm8</i>	MRI	V/i	AVX	Extract a qword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r64/m64</i> .

**NOTES:**

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXRB).

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

**Description**

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from *imm8[3:0]*. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXRB/PEXRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros. Attempt to execute VPEXTRQ in non-64-bit mode will cause #UD.

## Operation

CASE of

```

PEXTRB: SEL ← COUNT[3:0];
    TEMP ← (Src >> SEL*8) AND FFH;
    IF (DEST = Mem8)
        THEN
            Mem8 ← TEMP[7:0];
    ELSE IF (64-Bit Mode and 64-bit register selected)
        THEN
            R64[7:0] ← TEMP[7:0];
            r64[63:8] ← ZERO_FILL; };
    ELSE
        R32[7:0] ← TEMP[7:0];
        r32[31:8] ← ZERO_FILL; };
    FI;
PEXTRD:SEL ← COUNT[1:0];
    TEMP ← (Src >> SEL*32) AND FFFF_FFFFH;
    DEST ← TEMP;
PEXTRQ: SEL ← COUNT[0];
    TEMP ← (Src >> SEL*64);
    DEST ← TEMP;

```

EASC:

### (V)PEXTRTD/(V)PEXTRQ

```

IF (64-Bit Mode and 64-bit dest operand)
THEN
    Src_Offset ← Imm8[0]
    r64/m64 ← (Src >> Src_Offset * 64)
ELSE
    Src_Offset ← Imm8[1:0]
    r32/m32 ← ((Src >> Src_Offset *32) AND OFFFFFFFh);
FI

```

### (V)PEXTRB ( dest=m8)

```

SRC_Offset ← Imm8[3:0]
Mem8 ← (Src >> Src_Offset*8)

```

### (V)PEXTRB ( dest=reg)

```

IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[3:0]
    DEST[7:0] ← ((Src >> Src_Offset*8) AND OFFh)
    DEST[63:8] ← ZERO_FILL;
ELSE
    SRC_Offset ← Imm8[3:0];
    DEST[7:0] ← ((Src >> Src_Offset*8) AND OFFh);
    DEST[31:8] ← ZERO_FILL;
FI

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

PEXTRB:     int _mm_extract_epi8 (_m128i src, const int ndx);
PEXTRD:     int _mm_extract_epi32 (_m128i src, const int ndx);

```

PEXTRQ:        \_\_int64 \_mm\_extract\_epi64 (\_\_m128i src, const int ndx);

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 5; additionally

- #UD              If VEX.L = 1.
- If VEX.vvvv != 1111B.
- If VPEXTRQ in non-64-bit mode, VEX.W=1.

**PEXTRW—Extract Word**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 bit Mode Support</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
OF C5 /r ib <sup>1</sup> PEXTRW <i>reg, mm, imm8</i>	RMI	V/V	SSE	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>reg</i> , bits 15:0. The upper bits of r32 or r64 is zeroed.
66 OF C5 /r ib PEXTRW <i>reg, xmm, imm8</i>	RMI	V/V	SSE2	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to <i>reg</i> , bits 15:0. The upper bits of r32 or r64 is zeroed.
66 OF 3A 15 /r ib PEXTRW <i>reg/m16, xmm, imm8</i>	MRI	V/V	SSE4_1	Extract the word specified by <i>imm8</i> from <i>xmm</i> and copy it to lowest 16 bits of <i>reg</i> or <i>m16</i> . Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW <i>reg, xmm1, imm8</i>	RMI	V <sup>2</sup> /V	AVX	Extract the word specified by <i>imm8</i> from <i>xmm1</i> and move it to <i>reg</i> , 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 <i>reg/m16, xmm2, imm8</i>	MRI	V/V	AVX	Extract a word integer value from <i>xmm2</i> at the source word offset specified by <i>imm8</i> into <i>reg</i> or <i>m16</i> . The upper bits of r64/r32 is filled with zeros.

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

**Description**

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

## Operation

```

IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL * 16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
THEN
    FOR (PEXTRW instruction with 64-bit source operand)
        { SEL ← COUNT[1:0];
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r64[15:0] ← TEMP[15:0];
          r64[63:16] ← ZERO_FILL; }
    FOR (PEXTRW instruction with 128-bit source operand)
        { SEL ← COUNT[2:0];
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r64[15:0] ← TEMP[15:0];
          r64[63:16] ← ZERO_FILL; }
ELSE
    FOR (PEXTRW instruction with 64-bit source operand)
        { SEL ← COUNT[1:0];
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r32[15:0] ← TEMP[15:0];
          r32[31:16] ← ZERO_FILL; }
    FOR (PEXTRW instruction with 128-bit source operand)
        { SEL ← COUNT[2:0];
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r32[15:0] ← TEMP[15:0];
          r32[31:16] ← ZERO_FILL; }
Fl;
Fl;

```

**(V)PEXTRW ( dest=m16 )**  
 SRC\_Offset ← Imm8[2:0]  
 Mem16 ← (Src >> Src\_Offset\*16)

**(V)PEXTRW ( dest=reg )**  
 IF (64-Bit Mode )
 THEN
 SRC\_Offset ← Imm8[2:0]
 DEST[15:0] ← ((Src >> Src\_Offset\*16) AND OFFFFh)
 DEST[63:16] ← ZERO\_FILL;
 ELSE
 SRC\_Offset ← Imm8[2:0]
 DEST[15:0] ← ((Src >> Src\_Offset\*16) AND OFFFFh)
 DEST[31:16] ← ZERO\_FILL;
 Fl

## Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW: int \_mm\_extract\_pi16 (\_\_m64 a, int n)  
 PEXTRW: int \_mm\_extract\_epi16 ( \_\_m128i a, int imm)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Exceptions Type 5; additionally

#UD                    If VEX.L = 1.  
                       If VEX.vvvv != 1111B.

## PHADDW/PHADDD — Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 01 /r <sup>1</sup> PHADDW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to <i>mm1</i> .
66 OF 38 01 /r PHADDW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to <i>xmm1</i> .
OF 38 02 /r PHADDD <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to <i>mm1</i> .
66 OF 38 02 /r PHADDD <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 01 /r VPHADDW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Add 16-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 02 /r VPHADDD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Add 32-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 01 /r VPHADDW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack to <i>ymm1</i> .
VEX.NDS.256.66.0F38.WIG 02 /r VPHADDD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Add 32-bit signed integers horizontally, pack to <i>ymm1</i> .

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). (V)PHADDD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

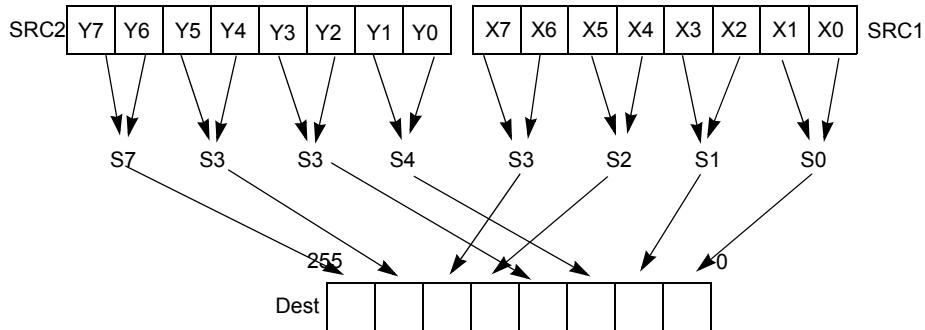
128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.



**Figure 4-6. 256-bit VPHADDD Instruction Operation**

## Operation

### PHADDW (with 64-bit operands)

$mm1[15:0] = mm1[31:16] + mm1[15:0];$   
 $mm1[31:16] = mm1[63:48] + mm1[47:32];$   
 $mm1[47:32] = mm2/m64[31:16] + mm2/m64[15:0];$   
 $mm1[63:48] = mm2/m64[63:48] + mm2/m64[47:32];$

### PHADDW (with 128-bit operands)

$xmm1[15:0] = xmm1[31:16] + xmm1[15:0];$   
 $xmm1[31:16] = xmm1[63:48] + xmm1[47:32];$   
 $xmm1[47:32] = xmm1[95:80] + xmm1[79:64];$   
 $xmm1[63:48] = xmm1[127:112] + xmm1[111:96];$   
 $xmm1[79:64] = xmm2/m128[31:16] + xmm2/m128[15:0];$   
 $xmm1[95:80] = xmm2/m128[63:48] + xmm2/m128[47:32];$   
 $xmm1[111:96] = xmm2/m128[95:80] + xmm2/m128[79:64];$   
 $xmm1[127:112] = xmm2/m128[127:112] + xmm2/m128[111:96];$

### VPHADDW (VEX.128 encoded version)

$DEST[15:0] \leftarrow SRC1[31:16] + SRC1[15:0]$   
 $DEST[31:16] \leftarrow SRC1[63:48] + SRC1[47:32]$   
 $DEST[47:32] \leftarrow SRC1[95:80] + SRC1[79:64]$   
 $DEST[63:48] \leftarrow SRC1[127:112] + SRC1[111:96]$   
 $DEST[79:64] \leftarrow SRC2[31:16] + SRC2[15:0]$   
 $DEST[95:80] \leftarrow SRC2[63:48] + SRC2[47:32]$   
 $DEST[111:96] \leftarrow SRC2[95:80] + SRC2[79:64]$   
 $DEST[127:112] \leftarrow SRC2[127:112] + SRC2[111:96]$   
 $DEST[VLMAX-1:128] \leftarrow 0$

**VPHADDW (VEX.256 encoded version)**

$\text{DEST}[15:0] \leftarrow \text{SRC1}[31:16] + \text{SRC1}[15:0]$   
 $\text{DEST}[31:16] \leftarrow \text{SRC1}[63:48] + \text{SRC1}[47:32]$   
 $\text{DEST}[47:32] \leftarrow \text{SRC1}[95:80] + \text{SRC1}[79:64]$   
 $\text{DEST}[63:48] \leftarrow \text{SRC1}[127:112] + \text{SRC1}[111:96]$   
 $\text{DEST}[79:64] \leftarrow \text{SRC2}[31:16] + \text{SRC2}[15:0]$   
 $\text{DEST}[95:80] \leftarrow \text{SRC2}[63:48] + \text{SRC2}[47:32]$   
 $\text{DEST}[111:96] \leftarrow \text{SRC2}[95:80] + \text{SRC2}[79:64]$   
 $\text{DEST}[127:112] \leftarrow \text{SRC2}[127:112] + \text{SRC2}[111:96]$   
 $\text{DEST}[143:128] \leftarrow \text{SRC1}[159:144] + \text{SRC1}[143:128]$   
 $\text{DEST}[159:144] \leftarrow \text{SRC1}[191:176] + \text{SRC1}[175:160]$   
 $\text{DEST}[175:160] \leftarrow \text{SRC1}[223:208] + \text{SRC1}[207:192]$   
 $\text{DEST}[191:176] \leftarrow \text{SRC1}[255:240] + \text{SRC1}[239:224]$   
 $\text{DEST}[207:192] \leftarrow \text{SRC2}[127:112] + \text{SRC2}[143:128]$   
 $\text{DEST}[223:208] \leftarrow \text{SRC2}[159:144] + \text{SRC2}[175:160]$   
 $\text{DEST}[239:224] \leftarrow \text{SRC2}[191:176] + \text{SRC2}[207:192]$   
 $\text{DEST}[255:240] \leftarrow \text{SRC2}[223:208] + \text{SRC2}[239:224]$

**PHADDD (with 64-bit operands)**

$\text{mm1}[31:0] = \text{mm1}[63:32] + \text{mm1}[31:0];$   
 $\text{mm1}[63:32] = \text{mm2/m64}[63:32] + \text{mm2/m64}[31:0];$

**PHADDD (with 128-bit operands)**

$\text{xmm1}[31:0] = \text{xmm1}[63:32] + \text{xmm1}[31:0];$   
 $\text{xmm1}[63:32] = \text{xmm1}[127:96] + \text{xmm1}[95:64];$   
 $\text{xmm1}[95:64] = \text{xmm2/m128}[63:32] + \text{xmm2/m128}[31:0];$   
 $\text{xmm1}[127:96] = \text{xmm2/m128}[127:96] + \text{xmm2/m128}[95:64];$

**VPHADDD (VEX.128 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VPHADDD (VEX.256 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[191:160] + \text{SRC1}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[255:224] + \text{SRC1}[223:192]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC2}[191:160] + \text{SRC2}[159:128]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[255:224] + \text{SRC2}[223:192]$

**Intel C/C++ Compiler Intrinsic Equivalents**

PHADDW:	<code>__m64 _mm_hadd_pi16 (__m64 a, __m64 b)</code>
PHADDD:	<code>__m64 _mm_hadd_pi32 (__m64 a, __m64 b)</code>
(V)PHADDW:	<code>__m128i _mm_hadd_epi16 (__m128i a, __m128i b)</code>
(V)PHADDD:	<code>__m128i _mm_hadd_epi32 (__m128i a, __m128i b)</code>
VPHADDW:	<code>__m256i _mm256_hadd_epi16 (__m256i a, __m256i b)</code>
VPHADDD:	<code>__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)</code>

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PHADDSW – Packed Horizontal Add and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 03 /r <sup>1</sup> PHADDSW mm1, mm2/m64	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to mm1.
66 0F 38 03 /r PHADDSW xmm1, xmm2/m128	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to xmm1.
VEX.NDS.128.66.0F38.WIG 03 /r VPHADDSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to xmm1.
VEX.NDS.256.66.0F38.WIG 03 /r VPHADDSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack saturated integers to ymm1.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PHADDSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PHADDSW (with 64-bit operands)

```

mm1[15-0] = SaturateToSignedWord((mm1[31-16] + mm1[15-0]);
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);

```

**PHADDSW (with 128-bit operands)**

```

xmm1[15:0] = SaturateToSignedWord(xmm1[31:16] + xmm1[15:0]);
xmm1[31:16] = SaturateToSignedWord(xmm1[63:48] + xmm1[47:32]);
xmm1[47:32] = SaturateToSignedWord(xmm1[95:80] + xmm1[79:64]);
xmm1[63:48] = SaturateToSignedWord(xmm1[127:112] + xmm1[111:96]);
xmm1[79:64] = SaturateToSignedWord(xmm2/m128[31:16] + xmm2/m128[15:0]);
xmm1[95:80] = SaturateToSignedWord(xmm2/m128[63:48] + xmm2/m128[47:32]);
xmm1[111:96] = SaturateToSignedWord(xmm2/m128[95:80] + xmm2/m128[79:64]);
xmm1[127:112] = SaturateToSignedWord(xmm2/m128[127:112] + xmm2/m128[111:96]);

```

**VPHADDSW (VEX.128 encoded version)**

```

DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[VLMAX-1:128] ← 0

```

**VPHADDSW (VEX.256 encoded version)**

```

DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[143:128] = SaturateToSignedWord(SRC1[159:144] + SRC1[143:128])
DEST[159:144] = SaturateToSignedWord(SRC1[191:176] + SRC1[175:160])
DEST[175:160] = SaturateToSignedWord(SRC1[223:208] + SRC1[207:192])
DEST[191:176] = SaturateToSignedWord(SRC1[255:240] + SRC1[239:224])
DEST[207:192] = SaturateToSignedWord(SRC2[127:112] + SRC2[143:128])
DEST[223:208] = SaturateToSignedWord(SRC2[159:144] + SRC2[175:160])
DEST[239:224] = SaturateToSignedWord(SRC2[191:160] + SRC2[159:128])
DEST[255:240] = SaturateToSignedWord(SRC2[255:240] + SRC2[239:224])

```

**Intel C/C++ Compiler Intrinsic Equivalent**

**PHADDSW:**    `__m64 _mm_hadds_pi16 (__m64 a, __m64 b)`  
**(V)PHADDSW:** `__m128i _mm_hadds_epi16 (__m128i a, __m128i b)`  
**VPHADDSW:**    `__m256i _mm256_hadds_epi16 (__m256i a, __m256i b)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                      If VEX.L = 1.

**PHMINPOSUW — Packed Horizontal Word Minimum**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 41 /r PHMINPOSUW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Find the minimum unsigned word in <i>xmm2/m128</i> and place its value in the low word of <i>xmm1</i> and its index in the second-lowest word of <i>xmm1</i> .
VEX.128.66.0F38.WIG 41 /r VPHMINPOSUW <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Find the minimum unsigned word in <i>xmm2/m128</i> and place its value in the low word of <i>xmm1</i> and its index in the second-lowest word of <i>xmm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first operand). The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

**Operation****PHMINPOSUW (128-bit Legacy SSE version)**

```

INDEX ← 0;
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN)
    THEN INDEX ← 1; MIN ← SRC[31:16]; Fl;
IF (SRC[47:32] < MIN)
    THEN INDEX ← 2; MIN ← SRC[47:32]; Fl;
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN)
    THEN INDEX ← 7; MIN ← SRC[127:112]; Fl;
DEST[15:0] ← MIN;
DEST[18:16] ← INDEX;
DEST[127:19] ← 00000000000000000000000000H;
```

**VPHMINPOSUW (VEX.128 encoded version)**

```

INDEX ← 0
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN) THEN INDEX ← 1; MIN ← SRC[31:16]
IF (SRC[47:32] < MIN) THEN INDEX ← 2; MIN ← SRC[47:32]
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN) THEN INDEX ← 7; MIN ← SRC[127:112]
DEST[15:0] ← MIN
DEST[18:16] ← INDEX
DEST[127:19] ← 0000000000000000000000000000H
DEST[VLMAX-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

`PHMINPOSUW:        __m128i _mm_minpos_epu16( __m128i packed_words);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

## PHSUBW/PHSUBD — Packed Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 05 /r <sup>1</sup> PHSUBW mm1, mm2/m64	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to mm1.
66 OF 38 05 /r PHSUBW xmm1, xmm2/m128	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to xmm1.
OF 38 06 /r PHSUBD mm1, mm2/m64	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to mm1.
66 OF 38 06 /r PHSUBD xmm1, xmm2/m128	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 05 /r VPHSUBW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 06 /r VPHSUBD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to xmm1.
VEX.NDS.256.66.0F38.WIG 05 /r VPHSUBW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract 16-bit signed integers horizontally, pack to ymm1.
VEX.NDS.256.66.0F38.WIG 06 /r VPHSUBD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract 32-bit signed integers horizontally, pack to ymm1.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

## Operation

### PHSUBW (with 64-bit operands)

```
mm1[15:0] = mm1[15:0] - mm1[31:16];
mm1[31:16] = mm1[47:32] - mm1[63:48];
mm1[47:32] = mm2/m64[15:0] - mm2/m64[31:16];
mm1[63:48] = mm2/m64[47:32] - mm2/m64[63:48];
```

### PHSUBW (with 128-bit operands)

```
xmm1[15:0] = xmm1[15:0] - xmm1[31:16];
xmm1[31:16] = xmm1[47:32] - xmm1[63:48];
xmm1[47:32] = xmm1[79:64] - xmm1[95:80];
xmm1[63:48] = xmm1[111:96] - xmm1[127:112];
xmm1[79:64] = xmm2/m128[15:0] - xmm2/m128[31:16];
xmm1[95:80] = xmm2/m128[47:32] - xmm2/m128[63:48];
xmm1[111:96] = xmm2/m128[79:64] - xmm2/m128[95:80];
xmm1[127:112] = xmm2/m128[111:96] - xmm2/m128[127:112];
```

### VPHSUBW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[VLMAX-1:128] ← 0
```

### VPHSUBW (VEX.256 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[143:128] ← SRC1[143:128] - SRC1[159:144]
DEST[159:144] ← SRC1[175:160] - SRC1[191:176]
DEST[175:160] ← SRC1[207:192] - SRC1[223:208]
DEST[191:176] ← SRC1[239:224] - SRC1[255:240]
DEST[207:192] ← SRC2[143:128] - SRC2[159:144]
DEST[223:208] ← SRC2[175:160] - SRC2[191:176]
DEST[239:224] ← SRC2[207:192] - SRC2[223:208]
DEST[255:240] ← SRC2[239:224] - SRC2[255:240]
```

### PHSUBD (with 64-bit operands)

```
mm1[31:0] = mm1[31:0] - mm1[63:32];
mm1[63:32] = mm2/m64[31:0] - mm2/m64[63:32];
```

**PHSUBD (with 128-bit operands)**

```

xmm1[31-0] = xmm1[31-0] - xmm1[63-32];
xmm1[63-32] = xmm1[95-64] - xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];

```

**VPHSUBD (VEX.128 encoded version)**

```

DEST[31-0] ← SRC1[31-0] - SRC1[63-32]
DEST[63-32] ← SRC1[95-64] - SRC1[127-96]
DEST[95-64] ← SRC2[31-0] - SRC2[63-32]
DEST[127-96] ← SRC2[95-64] - SRC2[127-96]
DEST[VLMAX-1:128] ← 0

```

**VPHSUBD (VEX.256 encoded version)**

```

DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC1[191:160]
DEST[191:160] ← SRC1[223:192] - SRC1[255:224]
DEST[223:192] ← SRC2[159:128] - SRC2[191:160]
DEST[255:224] ← SRC2[223:192] - SRC2[255:224]

```

**Intel C/C++ Compiler Intrinsic Equivalents**

PHSUBW:	<code>__m64 _mm_hsub_pi16 (__m64 a, __m64 b)</code>
PHSUBD:	<code>__m64 _mm_hsub_pi32 (__m64 a, __m64 b)</code>
(V)PHSUBW:	<code>__m128i _mm_hsub_epi16 (__m128i a, __m128i b)</code>
(V)PHSUBD:	<code>__m128i _mm_hsub_epi32 (__m128i a, __m128i b)</code>
VPHSUBW:	<code>__m256i _mm256_hsub_epi16 (__m256i a, __m256i b)</code>
VPHSUBD:	<code>__m256i _mm256_hsub_epi32 (__m256i a, __m256i b)</code>

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PHSUBSW – Packed Horizontal Subtract and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 07 /r <sup>1</sup> PHSUBSW mm1, mm2/m64	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to mm1.
66 0F 38 07 /r PHSUBSW xmm1, xmm2/m128	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.
VEX.NDS.128.66.0F38.WIG 07 /r VPHSUBSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.
VEX.NDS.256.66.0F38.WIG 07 /r VPHSUBSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract 16-bit signed integer horizontally, pack saturated integers to ymm1.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PHSUBSW (with 64-bit operands)

mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);  
 mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);  
 mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);  
 mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);

**PHSUBSW (with 128-bit operands)**

```

xmm1[15:0] = SaturateToSignedWord(xmm1[15:0] - xmm1[31:16]);
xmm1[31:16] = SaturateToSignedWord(xmm1[47:32] - xmm1[63:48]);
xmm1[47:32] = SaturateToSignedWord(xmm1[79:64] - xmm1[95:80]);
xmm1[63:48] = SaturateToSignedWord(xmm1[111:96] - xmm1[127:112]);
xmm1[79:64] = SaturateToSignedWord(xmm2/m128[15:0] - xmm2/m128[31:16]);
xmm1[95:80] = SaturateToSignedWord(xmm2/m128[47:32] - xmm2/m128[63:48]);
xmm1[111:96] = SaturateToSignedWord(xmm2/m128[79:64] - xmm2/m128[95:80]);
xmm1[127:112] = SaturateToSignedWord(xmm2/m128[111:96] - xmm2/m128[127:112]);

```

**VPHSUBSW (VEX.128 encoded version)**

```

DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[VLMAX-1:128] ← 0

```

**VPHSUBSW (VEX.256 encoded version)**

```

DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[143:128] = SaturateToSignedWord(SRC1[143:128] - SRC1[159:144])
DEST[159:144] = SaturateToSignedWord(SRC1[175:160] - SRC1[191:176])
DEST[175:160] = SaturateToSignedWord(SRC1[207:192] - SRC1[223:208])
DEST[191:176] = SaturateToSignedWord(SRC1[239:224] - SRC1[255:240])
DEST[207:192] = SaturateToSignedWord(SRC2[143:128] - SRC2[159:144])
DEST[223:208] = SaturateToSignedWord(SRC2[175:160] - SRC2[191:176])
DEST[239:224] = SaturateToSignedWord(SRC2[207:192] - SRC2[223:208])
DEST[255:240] = SaturateToSignedWord(SRC2[239:224] - SRC2[255:240])

```

**Intel C/C++ Compiler Intrinsic Equivalent**

PHSUBSW:	<code>_m64 _mm_hsubs_pi16 (_m64 a, _m64 b)</code>
(V)PHSUBSW:	<code>_m128i _mm_hsubs_епi16 (_m128i a, _m128i b)</code>
VPHSUBSW:	<code>_m256i _mm256_hsubs_епi16 (_m256i a, _m256i b)</code>

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

**PINSRB/PINSRD/PINSRQ – Insert Byte/Dword/Qword**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 3A 20 /r ib PINSRB <i>xmm1, r32/m8, imm8</i>	RMI	V/V	SSE4_1	Insert a byte integer value from <i>r32/m8</i> into <i>xmm1</i> at the destination element in <i>xmm1</i> specified by <i>imm8</i> .
66 OF 3A 22 /r ib PINSRD <i>xmm1, r/m32, imm8</i>	RMI	V/V	SSE4_1	Insert a dword integer value from <i>r/m32</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
66 REX.W OF 3A 22 /r ib PINSRQ <i>xmm1, r/m64, imm8</i>	RMI	V/N. E.	SSE4_1	Insert a qword integer value from <i>r/m64</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB <i>xmm1, xmm2, r32/m8, imm8</i>	RVMI	V <sup>1</sup> /V	AVX	Merge a byte integer value from <i>r32/m8</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the byte offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD <i>xmm1, xmm2, r/m32, imm8</i>	RVMI	V/V	AVX	Insert a dword integer value from <i>r32/m32</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the dword offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ <i>xmm1, xmm2, r/m64, imm8</i>	RVMI	V/I	AVX	Insert a qword integer value from <i>r64/m64</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the qword offset in <i>imm8</i> .

**NOTES:**

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

## Operation

CASE OF

```

PINSRB: SEL ← COUNT[3:0];
          MASK ← (0FFH << (SEL * 8));
          TEMP ← (((SRC[7:0] << (SEL * 8)) AND MASK);
PINSRD: SEL ← COUNT[1:0];
          MASK ← (0xFFFFFFFFH << (SEL * 32));
          TEMP ← (((SRC << (SEL * 32)) AND MASK) ;
PINSRQ: SEL ← COUNT[0]
          MASK ← (0xFFFFFFFFFFFFFFFH << (SEL * 64));
          TEMP ← (((SRC << (SEL * 32)) AND MASK) ;
ESAC;
          DEST ← ((DEST AND NOT MASK) OR TEMP);

```

### VPINSRB (VEX.128 encoded version)

```

SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0

```

### VPINSRD (VEX.128 encoded version)

```

SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0

```

### VPINSRQ (VEX.128 encoded version)

```

SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

PINSRB:     __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD:     __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ:     __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

```

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VPINSRQ in non-64-bit mode with VEX.W=1.

**PINSRW—Insert Word**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C4 /r ib <sup>1</sup> PINSRW <i>mm, r32/m16, imm8</i>	RMI	V/V	SSE	Insert the low word from <i>r32</i> or from <i>m16</i> into <i>mm</i> at the word position specified by <i>imm8</i> .
66 OF C4 /r ib PINSRW <i>xmm, r32/m16, imm8</i>	RMI	V/V	SSE2	Move the low word of <i>r32</i> or from <i>m16</i> into <i>xmm</i> at the word position specified by <i>imm8</i> .
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW <i>xmm1, xmm2, r32/m16, imm8</i>	RVMI	V <sup>2</sup> /V	AVX	Insert a word integer value from <i>r32/m16</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the word offset in <i>imm8</i> .

**NOTES:**

- See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.
- In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

**Operation****PINSRW (with 64-bit source operand)**

```

SEL ← COUNT AND 3H;
CASE (Determine word position) OF
  SEL ← 0: MASK ← 000000000000FFFFH;
  SEL ← 1: MASK ← 00000000FFFF0000H;
  SEL ← 2: MASK ← 0000FFFF00000000H;
  SEL ← 3: MASK ← FFFF000000000000H;
DEST ← (DEST AND NOT MASK) OR ((SRC << (SEL * 16)) AND MASK);

```

**PINSRW (with 128-bit source operand)**

```

SEL ← COUNT AND 7H;
CASE (Determine word position) OF
  SEL ← 0: MASK ← 00000000000000000000000000000000FFFFH;
  SEL ← 1: MASK ← 00000000000000000000000000000000FFFF0000H;
  SEL ← 2: MASK ← 00000000000000000000000000000000FFFF0000000H;
  SEL ← 3: MASK ← 00000000000000000000000000000000FFFF00000000H;
  SEL ← 4: MASK ← 0000000000000000FFFF0000000000000000000000H;
  SEL ← 5: MASK ← 00000000FFFF0000000000000000000000000000H;
  SEL ← 6: MASK ← 0000FFFF00000000000000000000000000000000H;
  SEL ← 7: MASK ← FFFF00000000000000000000000000000000000000H;
DEST ← (DEST AND NOT MASK) OR ((SRC << (SEL * 16)) AND MASK);

```

**VPINSRW (VEX.128 encoded version)**

```

SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

PINSRW:     __m64 _mm_insert_pi16 (__m64 a, int d, int n)
PINSRW:     __m128i _mm_insert_epi16 ( __m128i a, int b, int imm)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VPINSRW in non-64-bit mode with VEX.W=1.

## PMADDUBSW – Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 04 /r <sup>1</sup> PMADDUBSW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>mm1</i> .
66 0F 38 04 /r PMADDUBSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>ymm1</i> .

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PMADDUBSW (with 64 bit operands)

```

DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);

```

**PMADDUBSW (with 128 bit operands)**

```
DEST[15:0] = SaturateToSignedWord(SRC[15:8]* DEST[15:8]+SRC[7:0]*DEST[7:0]);
// Repeat operation for 2nd through 7th word
SRC1/DEST[127:112] = SaturateToSignedWord(SRC[127:120]*DEST[127:120]+ SRC[119:112]* DEST[119:112]);
```

**VPMADDUBSW (VEX.128 encoded version)**

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[VLMAX-1:128] ← 0
```

**VPMADDUBSW (VEX.256 encoded version)**

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
```

**Intel C/C++ Compiler Intrinsic Equivalents**

PMADDUBSW:	<code>__m64 _mm_maddubs_pi16 (__m64 a, __m64 b)</code>
(V)PMADDUBSW:	<code>__m128i _mm_maddubs_ep16 (__m128i a, __m128i b)</code>
VPMADDUBSW:	<code>__m256i _mm256_maddubs_ep16 (__m256i a, __m256i b)</code>

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

**PMADDWD—Multiply and Add Packed Integers**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF F5 /r <sup>1</sup> PMADDWD <i>mm, mm/m64</i>	RM	V/V	MMX	Multiply the packed words in <i>mm</i> by the packed words in <i>mm/m64</i> , add adjacent doubleword results, and store in <i>mm</i> .
66 OF F5 /r PMADDWD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed word integers in <i>xmm1</i> by the packed word integers in <i>xmm2/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed word integers in <i>xmm2</i> by the packed word integers in <i>xmm3/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG F5 /r VPMADDWD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed word integers in <i>ymm2</i> by the packed word integers in <i>ymm3/m256</i> , add adjacent doubleword results, and store in <i>ymm1</i> .

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-7 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

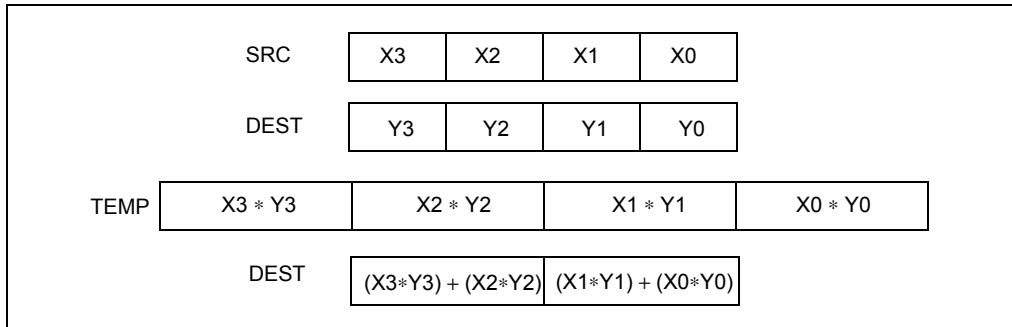
Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.



**Figure 4-7. PMADDWD Execution Model Using 64-bit Operands**

## Operation

### PMADDWD (with 64-bit operands)

```
DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);
```

### PMADDWD (with 128-bit operands)

```
DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);
DEST[95:64] ← (DEST[79:64] * SRC[79:64]) + (DEST[95:80] * SRC[95:80]);
DEST[127:96] ← (DEST[111:96] * SRC[111:96]) + (DEST[127:112] * SRC[127:112]);
```

### VPMADDWD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16]);
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48]);
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80]);
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

### VPMADDWD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16]);
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48]);
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80]);
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112]);
DEST[159:128] ← (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144]);
DEST[191:160] ← (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176]);
DEST[223:192] ← (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208]);
DEST[255:224] ← (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
```

## Intel C/C++ Compiler Intrinsic Equivalent

```
PMADDWD: __m64 _mm_madd_pi16(__m64 m1, __m64 m2)
(V)PMADDWD: __m128i _mm_madd_epi16 (__m128i a, __m128i b)
VPMADDWD: __m256i _mm256_madd_epi16 (__m256i a, __m256i b)
```

## Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PMAXSB – Maximum of Packed Signed Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3C /r PMAXSB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed maximum values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Compares packed signed byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[7:0] > SRC[7:0])
    THEN DEST[7:0] ← DEST[7:0];
    ELSE DEST[7:0] ← SRC[7:0]; Fl;
IF (DEST[15:8] > SRC[15:8])
    THEN DEST[15:8] ← DEST[15:8];
    ELSE DEST[15:8] ← SRC[15:8]; Fl;
IF (DEST[23:16] > SRC[23:16])
    THEN DEST[23:16] ← DEST[23:16];
    ELSE DEST[23:16] ← SRC[23:16]; Fl;
IF (DEST[31:24] > SRC[31:24])
    THEN DEST[31:24] ← DEST[31:24];
    ELSE DEST[31:24] ← SRC[31:24]; Fl;
IF (DEST[39:32] > SRC[39:32])
    THEN DEST[39:32] ← DEST[39:32];
    ELSE DEST[39:32] ← SRC[39:32]; Fl;
IF (DEST[47:40] > SRC[47:40])
    THEN DEST[47:40] ← DEST[47:40];

```

```

    ELSE DEST[47:40] ← SRC[47:40]; Fl;
IF (DEST[55:48] > SRC[55:48])
    THEN DEST[55:48] ← DEST[55:48];
    ELSE DEST[55:48] ← SRC[55:48]; Fl;
IF (DEST[63:56] > SRC[63:56])
    THEN DEST[63:56] ← DEST[63:56];
    ELSE DEST[63:56] ← SRC[63:56]; Fl;
IF (DEST[71:64] > SRC[71:64])
    THEN DEST[71:64] ← DEST[71:64];
    ELSE DEST[71:64] ← SRC[71:64]; Fl;
IF (DEST[79:72] > SRC[79:72])
    THEN DEST[79:72] ← DEST[79:72];
    ELSE DEST[79:72] ← SRC[79:72]; Fl;
IF (DEST[87:80] > SRC[87:80])
    THEN DEST[87:80] ← DEST[87:80];
    ELSE DEST[87:80] ← SRC[87:80]; Fl;
IF (DEST[95:88] > SRC[95:88])
    THEN DEST[95:88] ← DEST[95:88];
    ELSE DEST[95:88] ← SRC[95:88]; Fl;
IF (DEST[103:96] > SRC[103:96])
    THEN DEST[103:96] ← DEST[103:96];
    ELSE DEST[103:96] ← SRC[103:96]; Fl;
IF (DEST[111:104] > SRC[111:104])
    THEN DEST[111:104] ← DEST[111:104];
    ELSE DEST[111:104] ← SRC[111:104]; Fl;
IF (DEST[119:112] > SRC[119:112])
    THEN DEST[119:112] ← DEST[119:112];
    ELSE DEST[119:112] ← SRC[119:112]; Fl;
IF (DEST[127:120] > SRC[127:120])
    THEN DEST[127:120] ← DEST[127:120];
    ELSE DEST[127:120] ← SRC[127:120]; Fl;

```

**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]; Fl;
(* 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]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMAXSB (VEX.256 encoded version)**

```

IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; Fl;
(* 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMAXSB: `__m128i _mm_max_epi8 ( __m128i a, __m128i b);`  
VPMAXSB: `__m256i _mm256_max_epi8 ( __m256i a, __m256i b);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

**PMAXSD — Maximum of Packed Signed Dword Integers**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3D /r PMAXSD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed signed dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed dword integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3D /r VPMAXSD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed dword integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed maximum values in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Compares packed signed dword integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

**Operation**

```

IF (DEST[31:0] > SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; Fl;
IF (DEST[63:32] > SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; Fl;
IF (DEST[95:64] > SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; Fl;
IF (DEST[127:96] > SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; Fl;

```

**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]; Fl;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:95] > SRC2[127:95] THEN
    DEST[127:95] ← SRC1[127:95];
ELSE
    DEST[127:95] ← SRC2[127:95]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMAXSD (VEX.256 encoded version)**

```

IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; Fl;
(* 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

PMAXSD:     `__m128i _mm_max_epi32 ( __m128i a, __m128i b);`  
 VPMAXSD:    `__m256i _mm256_max_epi32 ( __m256i a, __m256i b);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

**PMAXSW—Maximum of Packed Signed Word Integers**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF EE /r <sup>1</sup> PMAXSW mm1, mm2/m64	RM	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return maximum values.
66 OF EE /r PMAXSW xmm1, xmm2/m128	RM	V/V	SSE2	Compare signed word integers in xmm2/m128 and xmm1 and return maximum values.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of word integers to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

**Operation****PMAXSW (64-bit operands)**

```
IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
```

(\* Repeat operation for 2nd and 3rd words in source and destination operands \*)

```

IF DEST[63:48] > SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; Fl;

```

**PMAXSW (128-bit operands)**

```

IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; Fl;
(* 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]; Fl;

```

**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]; Fl;
(* 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]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMAXSW (VEX.256 encoded version)**

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; Fl;
(* 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

**PMAXSW:**   `_m64 _mm_max_pi16(_m64 a, _m64 b)`  
**(V)PMAXSW:** `_m128i _mm_max_epi16 (_m128i a, _m128i b)`  
**VPMAXSW:** `_m256i _mm256_max_epi16 (_m256i a, _m256i b)`

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

## INSTRUCTION SET REFERENCE, N-Z

#UD                  If VEX.L = 1.

## PMAXUB—Maximum of Packed Unsigned Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF DE / <sup>1</sup> PMAXUB mm1, mm2/m64	RM	V/V	SSE	Compare unsigned byte integers in <i>mm2/m64</i> and <i>mm1</i> and returns maximum values.
66 OF DE /r PMAXUB xmm1, xmm2/m128	RM	V/V	SSE2	Compare unsigned byte integers in <i>xmm2/m128</i> and <i>xmm1</i> and returns maximum values.
VEX.NDS.128.66.0F.WIG DE /r VPMAXUB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG DE /r VPMAXUB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned byte integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed maximum values in <i>ymm1</i> .

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of byte integers to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PMAXUB (64-bit operands)

```
IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; Fl;
```

## INSTRUCTION SET REFERENCE, N-Z

```
(* 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]; Fl;
```

### PMAXUB (128-bit operands)

```
IF DEST[7:0] > SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; Fl;
(* 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]; Fl;
```

### 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]; Fl;
(* 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]; Fl;
DEST[VLMAX-1:128] ← 0
```

### VPMAXUB (VEX.256 encoded version)

```
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; Fl;
(* 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]; Fl;
```

### Intel C/C++ Compiler Intrinsic Equivalent

PMAXUB:       `__m64 _mm_max_pu8(__m64 a, __m64 b)`  
(V)PMAXUB:     `__m128i _mm_max_epu8 ( __m128i a, __m128i b)`  
VPMAXUB:       `__m256i _mm256_max_epu8 ( __m256i a, __m256i b);`

### Flags Affected

None.

### Numeric Exceptions

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PMAXUD — Maximum of Packed Unsigned Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed unsigned dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed unsigned dword integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3F /r VPMAXUD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed unsigned dword integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed maximum values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Compares packed unsigned dword integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[31:0] > SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; Fl;
IF (DEST[63:32] > SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; Fl;
IF (DEST[95:64] > SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; Fl;
IF (DEST[127:96] > SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; Fl;
```

### 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]; Fl;
(* Repeat operation for 2nd through 3rd dwds in source and destination operands *)
IF SRC1[127:95] > SRC2[127:95] THEN
    DEST[127:95] ← SRC1[127:95];
ELSE
    DEST[127:95] ← SRC2[127:95]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMAXUD (VEX.256 encoded version)**

```

IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; Fl;
(* Repeat operation for 2nd through 7th dwds 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMAXUD: `__m128i _mm_max_epu32 ( __m128i a, __m128i b);`  
 VPMAXUD: `__m256i _mm256_max_epu32 ( __m256i a, __m256i b);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PMAXUW – Maximum of Packed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3E /r PMAXUW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed unsigned word integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3E/r VPMAXUW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed unsigned word integers in <i>xmm3/m128</i> and <i>xmm2</i> and store maximum packed values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3E /r VPMAXUW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed unsigned word integers in <i>ymm3/m256</i> and <i>ymm2</i> and store maximum packed values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Compares packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[15:0] > SRC[15:0])
    THEN DEST[15:0] ← DEST[15:0];
    ELSE DEST[15:0] ← SRC[15:0]; Fl;
IF (DEST[31:16] > SRC[31:16])
    THEN DEST[31:16] ← DEST[31:16];
    ELSE DEST[31:16] ← SRC[31:16]; Fl;
IF (DEST[47:32] > SRC[47:32])
    THEN DEST[47:32] ← DEST[47:32];
    ELSE DEST[47:32] ← SRC[47:32]; Fl;
IF (DEST[63:48] > SRC[63:48])
    THEN DEST[63:48] ← DEST[63:48];
    ELSE DEST[63:48] ← SRC[63:48]; Fl;
IF (DEST[79:64] > SRC[79:64])
    THEN DEST[79:64] ← DEST[79:64];
    ELSE DEST[79:64] ← SRC[79:64]; Fl;
IF (DEST[95:80] > SRC[95:80])
    THEN DEST[95:80] ← DEST[95:80];

```

```

ELSE DEST[95:80] ← SRC[95:80]; Fl;
IF (DEST[111:96] > SRC[111:96])
    THEN DEST[111:96] ← DEST[111:96];
    ELSE DEST[111:96] ← SRC[111:96]; Fl;
IF (DEST[127:112] > SRC[127:112])
    THEN DEST[127:112] ← DEST[127:112];
    ELSE DEST[127:112] ← SRC[127:112]; Fl;

```

**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]; Fl;
(* 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]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMAXUW (VEX.256 encoded version)**

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; Fl;
(* 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMAXUW: `__m128i _mm_max_epu16 ( __m128i a, __m128i b);`  
 VPMAXUW: `__m256i _mm256_max_epu16 ( __m256i a, __m256i b)`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally  
#UD                    If VEX.L = 1.

## PMINSB — Minimum of Packed Signed Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 38 /r PMINSB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 38 /r VPMINSB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 38 /r VPMINSB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed minimum values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Compares packed signed byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[7:0] < SRC[7:0])
    THEN DEST[7:0] ← DEST[7:0];
    ELSE DEST[7:0] ← SRC[7:0]; Fl;
IF (DEST[15:8] < SRC[15:8])
    THEN DEST[15:8] ← DEST[15:8];
    ELSE DEST[15:8] ← SRC[15:8]; Fl;
IF (DEST[23:16] < SRC[23:16])
    THEN DEST[23:16] ← DEST[23:16];
    ELSE DEST[23:16] ← SRC[23:16]; Fl;
IF (DEST[31:24] < SRC[31:24])
    THEN DEST[31:24] ← DEST[31:24];
    ELSE DEST[31:24] ← SRC[31:24]; Fl;
IF (DEST[39:32] < SRC[39:32])
    THEN DEST[39:32] ← DEST[39:32];
    ELSE DEST[39:32] ← SRC[39:32]; Fl;
IF (DEST[47:40] < SRC[47:40])
    THEN DEST[47:40] ← DEST[47:40];

```

```

    ELSE DEST[47:40] ← SRC[47:40]; Fl;
IF (DEST[55:48] < SRC[55:48])
    THEN DEST[55:48] ← DEST[55:48];
    ELSE DEST[55:48] ← SRC[55:48]; Fl;
IF (DEST[63:56] < SRC[63:56])
    THEN DEST[63:56] ← DEST[63:56];
    ELSE DEST[63:56] ← SRC[63:56]; Fl;
IF (DEST[71:64] < SRC[71:64])
    THEN DEST[71:64] ← DEST[71:64];
    ELSE DEST[71:64] ← SRC[71:64]; Fl;
IF (DEST[79:72] < SRC[79:72])
    THEN DEST[79:72] ← DEST[79:72];
    ELSE DEST[79:72] ← SRC[79:72]; Fl;
IF (DEST[87:80] < SRC[87:80])
    THEN DEST[87:80] ← DEST[87:80];
    ELSE DEST[87:80] ← SRC[87:80]; Fl;
IF (DEST[95:88] < SRC[95:88])
    THEN DEST[95:88] ← DEST[95:88];
    ELSE DEST[95:88] ← SRC[95:88]; Fl;
IF (DEST[103:96] < SRC[103:96])
    THEN DEST[103:96] ← DEST[103:96];
    ELSE DEST[103:96] ← SRC[103:96]; Fl;
IF (DEST[111:104] < SRC[111:104])
    THEN DEST[111:104] ← DEST[111:104];
    ELSE DEST[111:104] ← SRC[111:104]; Fl;
IF (DEST[119:112] < SRC[119:112])
    THEN DEST[119:112] ← DEST[119:112];
    ELSE DEST[119:112] ← SRC[119:112]; Fl;
IF (DEST[127:120] < SRC[127:120])
    THEN DEST[127:120] ← DEST[127:120];
    ELSE DEST[127:120] ← SRC[127:120]; Fl;

```

**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]; Fl;
(* 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]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMINSB (VEX.256 encoded version)**

```

IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; Fl;
(* 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMINSB:    \_\_m128i \_mm\_min\_epi8 ( \_\_m128i a, \_\_m128i b);  
VPMINSB:    \_\_m256i \_mm256\_min\_epi8 ( \_\_m256i a, \_\_m256i b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

## PMINSD — Minimum of Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed signed dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed dword integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed dword integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed minimum values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Compares packed signed dword integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[31:0] < SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; Fl;
IF (DEST[63:32] < SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; Fl;
IF (DEST[95:64] < SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; Fl;
IF (DEST[127:96] < SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; Fl;
```

### 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]; Fl;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:95] < SRC2[127:95] THEN
    DEST[127:95] ← SRC1[127:95];
ELSE
    DEST[127:95] ← SRC2[127:95]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMINSD (VEX.256 encoded version)**

```

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; Fl;
(* 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMINSD:    \_\_m128i\_mm\_min\_epi32 ( \_\_m128i a, \_\_m128i b);  
 VPMINSD:    \_\_m256i\_mm256\_min\_epi32 ( \_\_m256i a, \_\_m256i b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

## PMINSW—Minimum of Packed Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF EA /r <sup>1</sup> PMINSW mm1, mm2/m64	RM	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return minimum values.
66 OF EA /r PMINSW xmm1, xmm2/m128	RM	V/V	SSE2	Compare signed word integers in xmm2/m128 and xmm1 and return minimum values.
VEX.NDS.128.66.0F.WIG EA /r VPMINSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F.WIG EA /r VPMINSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of word integers to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PMINSW (64-bit operands)

```

IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)

```

```
IF DEST[63:48] < SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; Fl;
```

**PMINSW (128-bit operands)**

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; Fl;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] < SRC/m64[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; Fl;
```

**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]; Fl;
(* 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]; Fl;
DEST[VLMAX-1:128] ← 0
```

**VPMINSW (VEX.256 encoded version)**

```
IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; Fl;
(* 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]; Fl;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

PMINSW: `__m64 _mm_min_pi16 (__m64 a, __m64 b)`  
(V)PMINSW: `__m128i _mm_min_epi16 ( __m128i a, __m128i b)`  
VPMINSW: `__m256i _mm256_min_epi16 ( __m256i a, __m256i b)`

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD

If VEX.L = 1.

#MF

(64-bit operations only) If there is a pending x87 FPU exception.

## PMINUB—Minimum of Packed Unsigned Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF DA /r <sup>1</sup> PMINUB mm1, mm2/m64	RM	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns minimum values.
66 OF DA /r PMINUB xmm1, xmm2/m128	RM	V/V	SSE2	Compare unsigned byte integers in xmm2/m128 and xmm1 and returns minimum values.
VEX.NDS.128.66.0F.WIG DA /r VPMINUB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F.WIG DA /r VPMINUB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of byte integers to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PMINUB (for 64-bit operands)

```

IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; Fl;

```

```
(* 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]; Fl;
```

**PMINUB (for 128-bit operands)**

```
IF DEST[7:0] < SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; Fl;
(* 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]; Fl;
```

**VPMINUB (VEX.128 encoded version)**

VPMINUB instruction for 128-bit operands:

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; Fl;
(* 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]; Fl;
DEST[VLMAX-1:128] ← 0
```

**VPMINUB (VEX.256 encoded version)**

VPMINUB instruction for 128-bit operands:

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; Fl;
(* 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]; Fl;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

PMINUB:	<code>__m64 _m_min_pu8 (__m64 a, __m64 b)</code>
(V)PMINUB:	<code>__m128i _mm_min_epu8 ( __m128i a, __m128i b)</code>
VPMINUB:	<code>__m256i _mm256_min_epu8 ( __m256i a, __m256i b)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PMINUD – Minimum of Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed unsigned dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed unsigned dword integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3B /r VPMINUD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed unsigned dword integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed minimum values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Compares packed unsigned dword integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[31:0] < SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; Fl;
IF (DEST[63:32] < SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; Fl;
IF (DEST[95:64] < SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; Fl;
IF (DEST[127:96] < SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; Fl;

```

### 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]; Fl;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:95] < SRC2[127:95] THEN
    DEST[127:95] ← SRC1[127:95];
ELSE
    DEST[127:95] ← SRC2[127:95]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMINUD (VEX.256 encoded version)**

VPMINUD instruction for 128-bit operands:

```

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; Fl;
(* 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMINUD:    \_\_m128i \_mm\_min\_epu32 ( \_\_m128i a, \_\_m128i b);  
 VPMINUD:    \_\_m256i \_mm256\_min\_epu32 ( \_\_m256i a, \_\_m256i b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

## PMINUW – Minimum of Packed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3A /r PMINUW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed unsigned word integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3A/r VPMINUW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed unsigned word integers in <i>xmm3/m128</i> and <i>xmm2</i> and return packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3A /r VPMINUW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed unsigned word integers in <i>ymm3/m256</i> and <i>ymm2</i> and return packed minimum values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Compares packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

IF (DEST[15:0] < SRC[15:0])
    THEN DEST[15:0] ← DEST[15:0];
    ELSE DEST[15:0] ← SRC[15:0]; Fl;
IF (DEST[31:16] < SRC[31:16])
    THEN DEST[31:16] ← DEST[31:16];
    ELSE DEST[31:16] ← SRC[31:16]; Fl;
IF (DEST[47:32] < SRC[47:32])
    THEN DEST[47:32] ← DEST[47:32];
    ELSE DEST[47:32] ← SRC[47:32]; Fl;
IF (DEST[63:48] < SRC[63:48])
    THEN DEST[63:48] ← DEST[63:48];
    ELSE DEST[63:48] ← SRC[63:48]; Fl;
IF (DEST[79:64] < SRC[79:64])
    THEN DEST[79:64] ← DEST[79:64];
    ELSE DEST[79:64] ← SRC[79:64]; Fl;
IF (DEST[95:80] < SRC[95:80])
    THEN DEST[95:80] ← DEST[95:80];

```

```

ELSE DEST[95:80] ← SRC[95:80]; Fl;
IF (DEST[111:96] < SRC[111:96])
    THEN DEST[111:96] ← DEST[111:96];
    ELSE DEST[111:96] ← SRC[111:96]; Fl;
IF (DEST[127:112] < SRC[127:112])
    THEN DEST[127:112] ← DEST[127:112];
    ELSE DEST[127:112] ← SRC[127:112]; Fl;

```

**VPMINUW (VEX.128 encoded version)**

VPMINUW instruction for 128-bit operands:

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; Fl;
(* 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]; Fl;
DEST[VLMAX-1:128] ← 0

```

**VPMINUW (VEX.256 encoded version)**

VPMINUW instruction for 128-bit operands:

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; Fl;
(* 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]; Fl;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMINUW:    \_\_m128i \_mm\_min\_epu16 ( \_\_m128i a, \_\_m128i b);  
 VPMINUW:    \_\_m256i \_mm256\_min\_epu16 ( \_\_m256i a, \_\_m256i b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                      If VEX.L = 1.

## PMOVMSKB—Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF D7 /r <sup>1</sup> PMOVMSKB reg, mm	RM	V/V	SSE	Move a byte mask of <i>mm</i> to <i>reg</i> . The upper bits of r32 or r64 are zeroed
66 OF D7 /r PMOVMSKB reg, xmm	RM	V/V	SSE2	Move a byte mask of <i>xmm</i> to <i>reg</i> . 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 <i>xmm1</i> to <i>reg</i> . 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 <i>ymm1</i> to <i>reg</i> . The upper bits of r64 are filled with zeros.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

Legacy SSE version: The source operand is an MMX technology register.

128-bit Legacy SSE version: The source operand is an XMM register.

VEX.128 encoded version: The source operand is an XMM register.

VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PMOVMSKB (with 64-bit source operand and r32)

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r32[7] ← SRC[63];
r32[31:8] ← ZERO_FILL;
```

#### (V)PMOVMSKB (with 128-bit source operand and r32)

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] ← SRC[127];
r32[31:16] ← ZERO_FILL;
```

**VPMOVMSKB (with 256-bit source operand and r32)**

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 3rd through 31*)
r32[31] ← SRC[255];
```

**PMOVMSKB (with 64-bit source operand and r64)**

```
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r64[7] ← SRC[63];
r64[63:8] ← ZERO_FILL;
```

**(V)PMOVMSKB (with 128-bit source operand and r64)**

```
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r64[15] ← SRC[127];
r64[63:16] ← ZERO_FILL;
```

**VPMOVMSKB (with 256-bit source operand and r64)**

```
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 31*)
r64[31] ← SRC[255];
r64[63:32] ← ZERO_FILL;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

PMOVMSKB:	int _mm_movemask_pi8(__m64 a)
(V)PMOVMSKB:	int _mm_movemask_epi8 ( __m128i a)
VPMOVMSKB:	int _mm256_movemask_epi8 ( __m256i a)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 7; additionally

#UD	If VEX.L = 1.
	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 <i>xmm1, xmm2/m64</i>	RM	V/V	SSE4_1	Sign extend 8 packed signed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed signed 16-bit integers in <i>xmm1</i> .
66 0f 38 21 /r PMOVSXBD <i>xmm1, xmm2/m32</i>	RM	V/V	SSE4_1	Sign extend 4 packed signed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed signed 32-bit integers in <i>xmm1</i> .
66 0f 38 22 /r  PMOVSXHQ <i>xmm1, xmm2/m16</i>	RM	V/V	SSE4_1	Sign extend 2 packed signed 8-bit integers in the low 2 bytes of <i>xmm2/m16</i> to 2 packed signed 64-bit integers in <i>xmm1</i> .
66 0f 38 23 /r PMOVSXWD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE4_1	Sign extend 4 packed signed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed signed 32-bit integers in <i>xmm1</i> .
66 0f 38 24 /r PMOVSXWQ <i>xmm1, xmm2/m32</i>	RM	V/V	SSE4_1	Sign extend 2 packed signed 16-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 2 packed signed 64-bit integers in <i>xmm1</i> .
66 0f 38 25 /r PMOVSDQ <i>xmm1, xmm2/m64</i>	RM	V/V	SSE4_1	Sign extend 2 packed signed 32-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 2 packed signed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 16-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD <i>xmm1, xmm2/m32</i>	RM	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 32-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 22 /r VPMOVSXHQ <i>xmm1, xmm2/m16</i>	RM	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of <i>xmm2/m16</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 32-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ <i>xmm1, xmm2/m32</i>	RM	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 25 /r VPMOVSDQ <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW <i>ymm1, xmm2/m128</i>	RM	V/V	AVX2	Sign extend 16 packed 8-bit integers in <i>xmm2/m128</i> to 16 packed 16-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD <i>ymm1, xmm2/m64</i>	RM	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 32-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 22 /r VPMOVSXHQ <i>ymm1, xmm2/m32</i>	RM	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 64-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD <i>ymm1, xmm2/m128</i>	RM	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 8 packed 32-bit integers in <i>ymm1</i> .

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ <i>ymm1, xmm2/m64</i>	RM	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 64-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 25 /r VPMOVSXHQ <i>ymm1, xmm2/m128</i>	RM	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 4 packed 64-bit integers in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Sign-extend the low byte/word/dword values in each word/dword/qword element of the source operand (second operand) to word/dword/qword integers and stored as packed data in the destination operand (first operand).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination register is YMM Register.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PMOVSBW

```
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]);
```

#### PMOVSBBD

```
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]);
```

#### PMOVSBQ

```
DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);
```

#### PMOVSWD

```
DEST[31:0] ← SignExtend(SRC[15:0]);
DEST[63:32] ← SignExtend(SRC[31:16]);
DEST[95:64] ← SignExtend(SRC[47:32]);
DEST[127:96] ← SignExtend(SRC[63:48]);
```

**PMOVSXWQ**

```
DEST[63:0] ← SignExtend(SRC[15:0]);
DEST[127:64] ← SignExtend(SRC[31:16]);
```

**PMOVSDQ**

```
DEST[63:0] ← SignExtend(SRC[31:0]);
DEST[127:64] ← SignExtend(SRC[63:32]);
```

**VPMOVSBW (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTEx\_to\_WORD()  
DEST[VLMAX-1:128] ← 0

**VPMOVSBXBD (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTEx\_to\_DWORD()  
DEST[VLMAX-1:128] ← 0

**VPMOVSBXHQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTEx\_to\_QWORD()  
DEST[VLMAX-1:128] ← 0

**VPMOVSXWD (VEX.128 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_DWORD()  
DEST[VLMAX-1:128] ← 0

**VPMOVSXWQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_QWORD()  
DEST[VLMAX-1:128] ← 0

**VPMOVSDQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD()  
DEST[VLMAX-1:128] ← 0

**VPMOVSBW (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTEx\_to\_WORD(DEST[127:0], SRC[63:0])  
Packed\_Sign\_Extend\_BYTEx\_to\_WORD(DEST[255:128], SRC[127:64])

**VPMOVSBXBD (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTEx\_to\_DWORD(DEST[127:0], SRC[31:0])  
Packed\_Sign\_Extend\_BYTEx\_to\_DWORD(DEST[255:128], SRC[63:32])

**VPMOVSBXHQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTEx\_to\_QWORD(DEST[127:0], SRC[15:0])  
Packed\_Sign\_Extend\_BYTEx\_to\_QWORD(DEST[255:128], SRC[31:16])

**VPMOVSXWD (VEX.256 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[63:0])  
Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[255:128], SRC[127:64])

**VPMOVSXWQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[31:0])  
Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[255:128], SRC[63:32])

**VPMOVVSXDQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[63:0])  
 Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[255:128], SRC[127:64])

**Intel C/C++ Compiler Intrinsic Equivalent**

```
(V)PMOVSXBW:    __m128i _mm_cvtepi8_epi16 ( __m128i a);
VPMOVSXBW:     __m256i _mm256_cvtepi8_epi16 ( __m128i a);
(V)PMOVSXBD:    __m128i _mm_cvtepi8_epi32 ( __m128i a);
VPMOVSXBD:     __m256i _mm256_cvtepi8_epi32 ( __m128i a);
(V)PMOVSXBQ:    __m128i _mm_cvtepi8_epi64 ( __m128i a);
VPMOVSXBQ:     __m256i _mm256_cvtepi8_epi64 ( __m128i a);
(V)PMOVSXWD:    __m128i _mm_cvtepi16_epi32 ( __m128i a);
VPMOVSXWD:     __m256i _mm256_cvtepi16_epi32 ( __m128i a);
(V)PMOVSXWQ:    __m128i _mm_cvtepi16_epi64 ( __m128i a);
VPMOVSXWQ:     __m256i _mm256_cvtepi16_epi64 ( __m128i a);
(V)PMOVSXDQ:    __m128i _mm_cvtepi32_epi64 ( __m128i a);
VPMOVSXDQ:     __m256i _mm256_cvtepi32_epi64 ( __m128i a);
```

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

## PMOVZX – Packed Move with Zero Extend

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW <i>xmm1, xmm2/m64</i>	RM	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 16-bit integers in <i>xmm1</i> .
66 0f 38 31 /r PMOVZXBD <i>xmm1, xmm2/m32</i>	RM	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 32-bit integers in <i>xmm1</i> .
66 0f 38 32 /r PMOVZXHQ <i>xmm1, xmm2/m16</i>	RM	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of <i>xmm2/m16</i> to 2 packed 64-bit integers in <i>xmm1</i> .
66 0f 38 33 /r PMOVZXWD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 32-bit integers in <i>xmm1</i> .
66 0f 38 34 /r PMOVZXWQ <i>xmm1, xmm2/m32</i>	RM	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 2 packed 64-bit integers in <i>xmm1</i> .
66 0f 38 35 /r PMOVZXDQ <i>xmm1, xmm2/m64</i>	RM	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 16-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 31 /r VPMOVZXBD <i>xmm1, xmm2/m32</i>	RM	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 32-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 32 /r VPMOVZXHQ <i>xmm1, xmm2/m16</i>	RM	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of <i>xmm2/m16</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 32-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ <i>xmm1, xmm2/m32</i>	RM	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 35 /r VPMOVZXDQ <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW <i>ymm1, xmm2/m128</i>	RM	V/V	AVX2	Zero extend 16 packed 8-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 16 packed 16-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD <i>ymm1, xmm2/m64</i>	RM	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 32-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 32 /r VPMOVZXHQ <i>ymm1, xmm2/m32</i>	RM	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 64-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD <i>ymm1, xmm2/m128</i>	RM	V/V	AVX2	Zero extend 8 packed 16-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 8 packed 32-bit integers in <i>ymm1</i> .

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ <i>y mm1, xmm2/m64</i>	RM	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 64-bit integers in <i>mm1</i> .
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ <i>y mm1, xmm2/m128</i>	RM	V/V	AVX2	Zero extend 4 packed 32-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 4 packed 64-bit integers in <i>mm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Zero-extend the low byte/word/dword values in each word/dword/qword element of the source operand (second operand) to word/dword/qword integers and stored as packed data in the destination operand (first operand).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination register is YMM Register.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PMOVZXBW

```
DEST[15:0] ← ZeroExtend(SRC[7:0]);
DEST[31:16] ← ZeroExtend(SRC[15:8]);
DEST[47:32] ← ZeroExtend(SRC[23:16]);
DEST[63:48] ← ZeroExtend(SRC[31:24]);
DEST[79:64] ← ZeroExtend(SRC[39:32]);
DEST[95:80] ← ZeroExtend(SRC[47:40]);
DEST[111:96] ← ZeroExtend(SRC[55:48]);
DEST[127:112] ← ZeroExtend(SRC[63:56]);
```

#### PMOVZXBD

```
DEST[31:0] ← ZeroExtend(SRC[7:0]);
DEST[63:32] ← ZeroExtend(SRC[15:8]);
DEST[95:64] ← ZeroExtend(SRC[23:16]);
DEST[127:96] ← ZeroExtend(SRC[31:24]);
```

#### PMOVZXQB

```
DEST[63:0] ← ZeroExtend(SRC[7:0]);
DEST[127:64] ← ZeroExtend(SRC[15:8]);
```

#### PMOVZXWD

```
DEST[31:0] ← ZeroExtend(SRC[15:0]);
DEST[63:32] ← ZeroExtend(SRC[31:16]);
DEST[95:64] ← ZeroExtend(SRC[47:32]);
DEST[127:96] ← ZeroExtend(SRC[63:48]);
```

**PMOVZXWQ**

$\text{DEST}[63:0] \leftarrow \text{ZeroExtend}(\text{SRC}[15:0]);$   
 $\text{DEST}[127:64] \leftarrow \text{ZeroExtend}(\text{SRC}[31:16]);$

**PMOVZXDQ**

$\text{DEST}[63:0] \leftarrow \text{ZeroExtend}(\text{SRC}[31:0]);$   
 $\text{DEST}[127:64] \leftarrow \text{ZeroExtend}(\text{SRC}[63:32]);$

**VPMOVZXBW (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTEToWORD()  
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VPMOVZXBD (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTEToDWORD()  
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VPMOVZXBQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTEToQWORD()  
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VPMOVZXWD (VEX.128 encoded version)**

Packed\_Zero\_Extend\_WORDToDWORD()  
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VPMOVZXWQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_WORDToQWORD()  
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VPMOVZXDQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_DWORDToQWORD()  
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VPMOVZXBW (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTEToWORD(DEST[127:0], SRC[63:0])  
Packed\_Zero\_Extend\_BYTEToWORD(DEST[255:128], SRC[127:64])

**VPMOVZXBD (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTEToDWORD(DEST[127:0], SRC[31:0])  
Packed\_Zero\_Extend\_BYTEToDWORD(DEST[255:128], SRC[63:32])

**VPMOVZXBQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTEToQWORD(DEST[127:0], SRC[15:0])  
Packed\_Zero\_Extend\_BYTEToQWORD(DEST[255:128], SRC[31:16])

**VPMOVZXWD (VEX.256 encoded version)**

Packed\_Zero\_Extend\_WORDToDWORD(DEST[127:0], SRC[63:0])  
Packed\_Zero\_Extend\_WORDToDWORD(DEST[255:128], SRC[127:64])

**VPMOVZXWQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_WORDToQWORD(DEST[127:0], SRC[31:0])  
Packed\_Zero\_Extend\_WORDToQWORD(DEST[255:128], SRC[63:32])

**VPMOVZXDQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[63:0])  
 Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST[255:128], SRC[127:64])

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

```
(V)PMOVZXBW: __m128i_mm_cvtepu8_epi16 ( __m128i a);
VPMOVZXBW: __m256i_mm256_cvtepu8_epi16 ( __m128i a);
(V)PMOVZXBD: __m128i_mm_cvtepu8_epi32 ( __m128i a);
VPMOVZXBD: __m256i_mm256_cvtepu8_epi32 ( __m128i a);
(V)PMOVZXHQ: __m128i_mm_cvtepu8_epi64 ( __m128i a);
VPMOVZXHQ: __m256i_mm256_cvtepu8_epi64 ( __m128i a);
(V)PMOVZXWD: __m128i_mm_cvtepu16_epi32 ( __m128i a);
VPMOVZXWD: __m256i_mm256_cvtepu16_epi32 ( __m128i a);
(V)PMOVZXWQ: __m128i_mm_cvtepu16_epi64 ( __m128i a);
VPMOVZXWQ: __m256i_mm256_cvtepu16_epi64 ( __m128i a);
(V)PMOVZXDQ: __m128i_mm_cvtepu32_epi64 ( __m128i a);
VPMOVZXDQ: __m256i_mm256_cvtepu32_epi64 ( __m128i a);
```

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5; additionally

#UD                  If VEX.L = 1.  
                     If VEX.vvvv != 1111B.

## PMULDDQ – Multiply Packed Signed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDDQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Multiply the packed signed dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store the quadword product in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDDQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply packed signed doubleword integers in <i>xmm2</i> by packed signed doubleword integers in <i>xmm3/m128</i> , and store the quadword results in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 28 /r VPMULDDQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply packed signed doubleword integers in <i>ymm2</i> by packed signed doubleword integers in <i>ymm3/m256</i> , and store the quadword results in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the first source operand by the second source operand and stores the result in the destination operand.

For PMULDQ and VPMULDQ (VEX.128 encoded version), the second source operand is two packed signed doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. The first source operand is two packed signed doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed signed quadword integers stored in an XMM register. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

For VPMULDQ (VEX.256 encoded version), the second source operand is four packed signed doubleword integers stored in the first (low), third, fifth and seventh doublewords of an YMM register or a 256-bit memory location. The first source operand is four packed signed doubleword integers stored in the first, third, fifth and seventh doublewords of an XMM register. The destination contains four packed signed quadword integers stored in an YMM register. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation.

When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

### Operation

#### PMULDQ (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[31:0] * SRC[31:0]
DEST[127:64] ← DEST[95:64] * SRC[95:64]
DEST[VLMAX-1:128] (Unmodified)
```

**VPMULDQ (VEX.128 encoded version)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VPMULDQ (VEX.256 encoded version)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$   
 $\text{DEST}[191:128] \leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128]$   
 $\text{DEST}[255:192] \leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192]$

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMULDQ:    `__m128i _mm_mul_ep32( __m128i a, __m128i b);`  
 VPMULDQ:    `__m256i _mm256_mul_ep32( __m256i a, __m256i b);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5; additionally

#UD                      If VEX.L = 1.  
                           If VEX.vvvv != 1111B.

## PMULHRSW – Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 0B /r <sup>1</sup> PMULHRSW mm1, mm2/m64	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to mm1.
66 OF 38 0B /r PMULHRSW xmm1, xmm2/m128	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Legacy SSE version: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

## Operation

### PMULHRSW (with 64-bit operands)

```

temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >> 14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >> 14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];

```

### PMULHRSW (with 128-bit operand)

```

temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112]) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
DEST[79:64] = temp4[16:1];
DEST[95:80] = temp5[16:1];
DEST[111:96] = temp6[16:1];
DEST[127:112] = temp7[16:1];

```

### VPMULHRSW (VEX.128 encoded version)

```

temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[VLMAX-1:128] ← 0

```

### VPMULHRSW (VEX.256 encoded version)

```

temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1

```

```

temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1
temp8[31:0] ← INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] ← INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] ← INT32 ((SRC1[75:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] ← INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] ← INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] ← INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] ← INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] ← INT32 ((SRC1[255:240] * SRC2[255:240]) >>14) + 1

```

### Intel C/C++ Compiler Intrinsic Equivalents

PMULHRSW:	<code>__m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)</code>
(V)PMULHRSW:	<code>__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)</code>
VPMULHRSW:	<code>__m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)</code>

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

**PMULHUW—Multiply Packed Unsigned Integers and Store High Result**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF E4 /r <sup>1</sup> PMULHUW <i>mm1, mm2/m64</i>	RM	V/V	SSE	Multiply the packed unsigned word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the high 16 bits of the results in <i>mm1</i> .
66 OF E4 /r PMULHUW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed unsigned word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG E4 /r VPMULHUW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed unsigned word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG E4 /r VPMULHUW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed unsigned word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the high 16 bits of the results in <i>ymm1</i> .

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

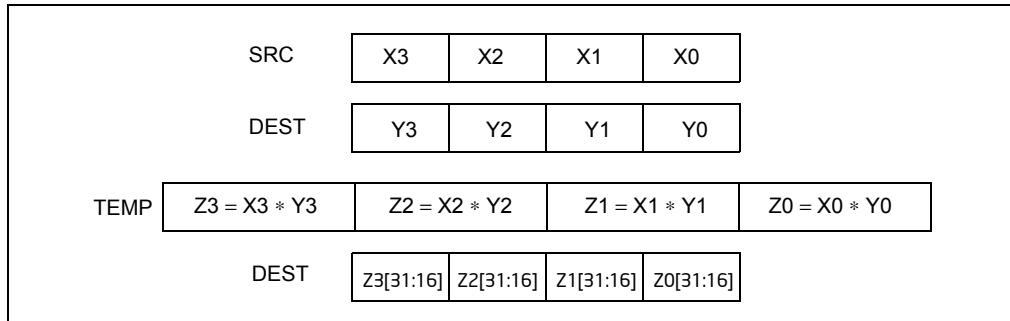


Figure 4-8. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

## Operation

### PMULHUW (with 64-bit operands)

```

TEMPO[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] ← TEMPO[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)

```

TEMPO[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] ← TEMPO[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)

```

TEMPO[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] ← TEMPO[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]

```

```

DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[VLMAX-1:128] ← 0

```

**PMULHUW (VEX.256 encoded version)**

```

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]

```

**Intel C/C++ Compiler Intrinsic Equivalent**

PMULHUW:	<code>_m64 _mm_mulhi_pu16(_m64 a, _m64 b)</code>
(V)PMULHUW:	<code>_m128i _mm_mulhi_epu16 (_m128i a, _m128i b)</code>
VPMULHUW:	<code>_m256i _mm256_mulhi_epu16 ( __m256i a, __m256i b)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF E5 /r <sup>1</sup> PMULHW mm, mm/m64	RM	V/V	MMX	Multiply the packed signed word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 OF E5 /r PMULHW xmm1, xmm2/m128	RM	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.128.66.0F.WIG E5 /r VPMULHW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.256.66.0F.WIG E5 /r VPMULHW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

### Operation

#### PMULHW (with 64-bit operands)

```

TEMPO[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] ← TEMPO[31:16];

```

```
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
```

**PMULHW (with 128-bit operands)**

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];
```

**VPMULHW (VEX.128 encoded version)**

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[VLMAX-1:128] ← 0
```

**PMULHW (VEX.256 encoded version)**

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
```

```

TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]

```

### Intel C/C++ Compiler Intrinsic Equivalent

PMULHW:    \_\_m64 \_mm\_mulhi\_pi16 (\_\_m64 m1, \_\_m64 m2)  
 (V)PMULHW: \_\_m128i \_mm\_mulhi\_epi16 ( \_\_m128i a, \_\_m128i b)  
 VPMULHW:   \_\_m256i \_mm256\_mulhi\_epi16 ( \_\_m256i a, \_\_m256i b)

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

## PMULLD – Multiply Packed Signed Dword Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Multiply the packed dword signed integers in <i>xmm1</i> and <i>xmm2/m128</i> and store the low 32 bits of each product in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 40 /r VPMULLD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed dword signed integers in <i>xmm2</i> and <i>xmm3/m128</i> and store the low 32 bits of each product in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 40 /r VPMULLD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed dword signed integers in <i>ymm2</i> and <i>ymm3/m256</i> and store the low 32 bits of each product in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs four signed multiplications from four pairs of signed dword integers and stores the lower 32 bits of the four 64-bit products in the destination operand (first operand). Each dword element in the destination operand is multiplied with the corresponding dword element of the source operand (second operand) to obtain a 64-bit intermediate product.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

```

Temp0[63:0] ← DEST[31:0] * SRC[31:0];
Temp1[63:0] ← DEST[63:32] * SRC[63:32];
Temp2[63:0] ← DEST[95:64] * SRC[95:64];
Temp3[63:0] ← DEST[127:96] * SRC[127:96];
DEST[31:0] ← Temp0[31:0];
DEST[63:32] ← Temp1[31:0];
DEST[95:64] ← Temp2[31:0];
DEST[127:96] ← Temp3[31:0];

```

### VPMULLD (VEX.128 encoded version)

```

Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
DEST[31:0] ← Temp0[31:0]

```

```
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[VLMAX-1:128] ← 0
```

**VPMULLD (VEX.256 encoded version)**

```
Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
Temp4[63:0] ← SRC1[159:128] * SRC2[159:128]
Temp5[63:0] ← SRC1[191:160] * SRC2[191:160]
Temp6[63:0] ← SRC1[223:192] * SRC2[223:192]
Temp7[63:0] ← SRC1[255:224] * SRC2[255:224]
```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PMULLUD: `__m128i _mm_mullo_epi32(__m128i a, __m128i b);`  
 VPMULLD: `__m256i _mm256_mullo_epi32(__m256i a, __m256i b);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF D5 / <sup>1</sup> PMULLW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Multiply the packed signed word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the low 16 bits of the results in <i>mm1</i> .
66 OF D5 / <i>r</i> PMULLW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the low 16 bits of the results in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG D5 / <i>r</i> VPMULLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed dword signed integers in <i>xmm2</i> and <i>xmm3/m128</i> and store the low 32 bits of each product in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG D5 / <i>r</i> VPMULLW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the low 16 bits of the results in <i>ymm1</i> .

### NOTES:

- See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

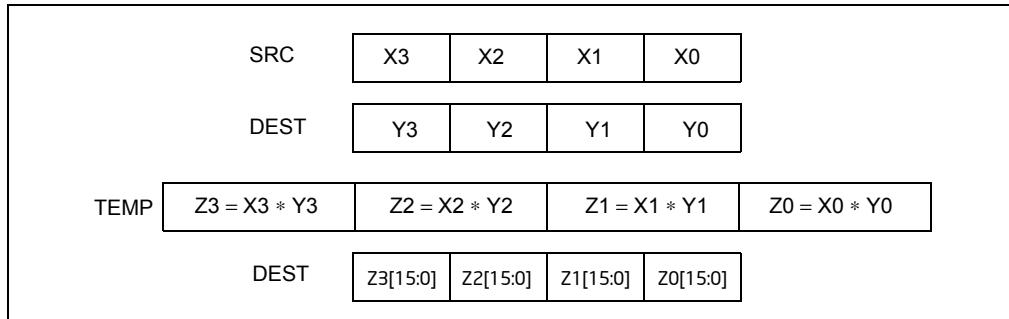


Figure 4-9. PMULLU Instruction Operation Using 64-bit Operands

## Operation

### PMULLW (with 64-bit operands)

```

TEMPO[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] ← TEMPO[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)

```

TEMPO[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] ← TEMPO[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];
DEST[79:64] ← TEMP4[15:0];
DEST[95:80] ← TEMP5[15:0];
DEST[111:96] ← TEMP6[15:0];
DEST[127:112] ← TEMP7[15:0];

```

### VPMULLW (VEX.128 encoded version)

```

Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]

```

```

DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[VLMAX-1:128] ← 0

```

**VPMULLD (VEX.256 encoded version)**

```

Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
Temp4[63:0] ← SRC1[159:128] * SRC2[159:128]
Temp5[63:0] ← SRC1[191:160] * SRC2[191:160]
Temp6[63:0] ← SRC1[223:192] * SRC2[223:192]
Temp7[63:0] ← SRC1[255:224] * SRC2[255:224]

```

```

DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[159:128] ← Temp4[31:0]
DEST[191:160] ← Temp5[31:0]
DEST[223:192] ← Temp6[31:0]
DEST[255:224] ← Temp7[31:0]

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

PMULLW:      __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
(V)PMULLW:   __m128i _mm_mullo_epi16 ( __m128i a, __m128i b)
VPMULLW:     __m256i _mm256_mullo_epi16 ( __m256i a, __m256i b);

```

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                  If VEX.L = 1.

## PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F4 /r <sup>1</sup> PMULUDQ mm1, mm2/m64	RM	V/V	SSE2	Multiply unsigned doubleword integer in <i>mm1</i> by unsigned doubleword integer in <i>mm2/m64</i> , and store the quadword result in <i>mm1</i> .
66 0F F4 /r PMULUDQ xmm1, xmm2/m128	RM	V/V	SSE2	Multiply packed unsigned doubleword integers in <i>xmm1</i> by packed unsigned doubleword integers in <i>xmm2/m128</i> , and store the quadword results in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed unsigned doubleword integers in <i>xmm2</i> by packed unsigned doubleword integers in <i>xmm3/m128</i> , and store the quadword results in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply packed unsigned doubleword integers in <i>ymm2</i> by packed unsigned doubleword integers in <i>ymm3/m256</i> , and store the quadword results in <i>ymm1</i> .

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE version:** The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register. The result is an unsigned quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

**128-bit Legacy SSE version:** The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded version:** The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first

source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

## Operation

### PMULUDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[31:0] * SRC[31:0];
```

### PMULUDQ (with 128-Bit operands)

```
DEST[63:0] ← DEST[31:0] * SRC[31:0];
```

```
DEST[127:64] ← DEST[95:64] * SRC[95:64];
```

### VPMULUDQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
```

```
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
```

```
DEST[VLMAX-1:128] ← 0
```

### VPMULUDQ (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
```

```
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
```

```
DEST[191:128] ← SRC1[159:128] * SRC2[159:128]
```

```
DEST[255:192] ← SRC1[223:192] * SRC2[223:192]
```

## Intel C/C++ Compiler Intrinsic Equivalent

PMULUDQ: `__m64 _mm_mul_su32 (__m64 a, __m64 b)`

(V)PMULUDQ: `__m128i _mm_mul_epu32 ( __m128i a, __m128i b)`

VPMULUDQ: `__m256i _mm256_mul_epu32( __m256i a, __m256i b);`

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## POP—Pop a Value from the Stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP r/m16	M	Valid	Valid	Pop top of stack into <i>m16</i> ; increment stack pointer.
8F /0	POP r/m32	M	N.E.	Valid	Pop top of stack into <i>m32</i> ; increment stack pointer.
8F /0	POP r/m64	M	Valid	N.E.	Pop top of stack into <i>m64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
58+ rw	POP r16	O	Valid	Valid	Pop top of stack into <i>r16</i> ; increment stack pointer.
58+ rd	POP r32	O	N.E.	Valid	Pop top of stack into <i>r32</i> ; increment stack pointer.
58+ rd	POP r64	O	Valid	N.E.	Pop top of stack into <i>r64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	NP	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	NP	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	NP	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
OF A1	POP FS	NP	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
OF A1	POP FS	NP	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
OF A1	POP FS	NP	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
OF A9	POP GS	NP	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
OF A9	POP GS	NP	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
OF A9	POP GS	NP	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA
O	opcode + rd (w)	NA	NA	NA
NP	NA	NA	NA	NA

### Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

The address size is used only when writing to a destination operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A NULL value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is NULL.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to OH as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt<sup>1</sup>. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POPs using 32-bit operands are not encodable and POPs to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF StackAddrSize = 32
  THEN
    IF OperandSize = 32
      THEN
        DEST ← SS:ESP; (* Copy a doubleword *)
        ESP ← ESP + 4;
      ELSE (* OperandSize = 16*)
        DEST ← SS:ESP; (* Copy a word *)
    
```

- 
1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a POP SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that POP the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

In the following sequence, interrupts may be recognized before POP ESP executes:

```
POP SS
POP SS
POP ESP
```

## INSTRUCTION SET REFERENCE, N-Z

```
    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-NUL selector;  
THEN  
    IF segment selector index is outside descriptor table limits  
    OR segment is not a data or readable code segment  
    OR ((segment is a data or nonconforming code segment)  
        AND (both RPL and CPL > DPL))  
        THEN #GP(selector);  
    IF segment not marked present  
        THEN #NP(selector);  
ELSE  
    SegmentRegister ← segment selector;  
    SegmentRegister ← segment descriptor;  
FI;  
FI;  
IF FS, or GS is loaded with a NUL selector;  
THEN  
    SegmentRegister ← segment selector;  
    SegmentRegister ← segment descriptor;  
FI;
```

PROTECTED 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-NUL selector;
THEN
  IF segment selector index is outside descriptor table limits
    or segment is not a data or readable code segment
    or ((segment is a data or nonconforming code segment)
        and (both RPL and CPL > DPL))
    THEN #GP(selector);
  FI;
  IF segment not marked present
    THEN #NP(selector);
  ELSE
    SegmentRegister ← segment selector;
    SegmentRegister ← segment descriptor;
  FI;
FI;

IF DS, ES, FS, or GS is loaded with a NUL 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 NUL 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 NUL segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits. If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.

	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(U)	If the stack address is in a non-canonical form.
#GP(selector)	If the descriptor is outside the descriptor table limit.
	If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code)	If a page fault occurs.
#NP	If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD	If the LOCK prefix is used.

## POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
61	POPA		NP	Invalid	Pop DI, SI, BP, BX, DX, CX, and AX.
61	POPAD		NP	Invalid	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment's segment descriptor determines the operand-size attribute.)

This instruction executes as described in non-64-bit modes. It is not valid in 64-bit mode.

### Operation

```

IF 64-Bit Mode
  THEN
    #UD;
ELSE
  IF OperandSize = 32 (* Instruction = POPAD *)
    THEN
      EDI ← Pop();
      ESI ← Pop();
      EBP ← Pop();
      Increment ESP by 4; (* Skip next 4 bytes of stack *)
      EBX ← Pop();
      EDX ← Pop();
      ECX ← Pop();
      EAX ← Pop();
    ELSE (* OperandSize = 16, instruction = POPA *)
      DI ← Pop();
      SI ← Pop();
      BP ← Pop();
      Increment ESP by 2; (* Skip next 2 bytes of stack *)
      BX ← Pop();
      DX ← Pop();
      CX ← Pop();
      AX ← Pop();
    FI;
FI;
```

**Flags Affected**

None.

**Protected Mode Exceptions**

- #SS(0) If the starting or ending stack address is not within the stack segment.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #SS If the starting or ending stack address is not within the stack segment.
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- #SS(0) If the starting or ending stack address is not within the stack segment.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If an unaligned memory reference is made while alignment checking is enabled.
- #UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

- #UD If in 64-bit mode.

## POPCNT – Return the Count of Number of Bits Set to 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 OF B8 /r	POPCNT r16, r/m16	RM	Valid	Valid	POPCNT on r/m16
F3 OF B8 /r	POPCNT r32, r/m32	RM	Valid	Valid	POPCNT on r/m32
F3 REX.W OF B8 /r	POPCNT r64, r/m64	RM	Valid	N.E.	POPCNT on r/m64

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction calculates of number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

### Operation

```
Count = 0;
For (i=0; i < OperandSize; i++)
{
    IF (SRC[ i ] = 1) // i'th bit
        THEN Count++; FI;
}
DEST ← Count;
```

### Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared

### Intel C/C++ Compiler Intrinsic Equivalent

```
POPCNT:     int _mm_popcnt_u32(unsigned int a);
POPCNT:     int64_t _mm_popcnt_u64(unsigned __int64 a);
```

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0. If LOCK prefix is used. Either the prefix REP (F3h) or REPN (F2H) is used.

### Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to OFFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0. If LOCK prefix is used. Either the prefix REP (F3h) or REPN (F2H) is used.

**Virtual 8086 Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to OFFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0. If LOCK prefix is used. Either the prefix REP (F3h) or REPN (F2H) is used.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0. If LOCK prefix is used. Either the prefix REP (F3h) or REPN (F2H) is used.

## POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description	
9D	POPF		NP	Valid	Pop top of stack into lower 16 bits of EFLAGS.	
9D	POPFD			NP	N.E.	Pop top of stack into EFLAGS.
9D	POPFQ			NP	Valid	Pop top of stack and zero-extend into RFLAGS.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. When the processor is operating in protected 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<sup>1</sup>, VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

When operating in protected 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 VIP, VIF, and VM. Here, the IOPL flags are unaffected, the VIP and VIF flags are cleared, and the VM flag is unaffected. 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, the IOPL must be equal to 3 to use POPF/POPFD instructions; VM, RF, IOPL, VIP, and VIF are unaffected. If the IOPL is less than 3, POPF/POPFD causes a general-protection exception (#GP).

In 64-bit mode, use REX.W to pop the top of stack to RFLAGS. The mnemonic assigned is POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64 bits from the stack, loads the lower 32 bits into RFLAGS, and zero extends the upper bits of RFLAGS.

See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the EFLAGS registers.

### Operation

```

IF VM = 0 (* Not in Virtual-8086 Mode *)
  THEN IF CPL = 0
    THEN
      IF OperandSize = 32;
        THEN
          EFLAGS ← Pop(); (* 32-bit pop *)
          (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
             VIP and VIF are cleared; RF, VM, and all reserved bits are unaffected. *)
      ELSE IF (Operandsize = 64)
        THEN
          RFLAGS ← Pop(); (* 64-bit pop *)
          (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
             VIP and VIF are cleared; RF, VM, and all reserved bits are unaffected. *)

```

1. RF is always zero after the execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.

```

RFLAGS = Pop(); (* 64-bit pop *)
(* All non-reserved flags except RF, VIP, VIF, and VM can be modified; VIP
and VIF are cleared; RF, VM, and all reserved bits are unaffected.*)
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, RF, VIP, and
VIF can be modified; IF, IOPL, RF, VM, and all reserved
bits are unaffected; VIP and VIF are cleared. *)
                ELSE
                    EFLAGS ← Pop(); (* 32-bit pop *)
                    (* All non-reserved bits except IOPL, RF, VIP, and VIF can be
modified; IOPL, RF, VM, and all reserved bits are
unaffected; VIP and VIF are cleared. *)
                FI;
            ELSE IF (Operandsize = 64)
                IF CPL > IOPL
                    THEN
                        RFLAGS ← Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IF, IOPL, RF, VIP, and
VIF can be modified; IF, IOPL, RF, VM, and all reserved
bits are unaffected; VIP and VIF are cleared. *)
                    ELSE
                        RFLAGS ← Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IOPL, RF, VIP, and VIF can be
modified; IOPL, RF, VM, and all reserved bits are
unaffected; VIP and VIF are 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 VM, RF, IOPL, VIP, and VIF can be
modified; VM, RF, IOPL, VIP, VIF, and all reserved bits are unaffected. *)
                    ELSE
                        EFLAGS[15:0] ← Pop(); FI;
                        (* All non-reserved bits except IOPL can be modified;
IOPL and all reserved bits are unaffected. *)
                ELSE (* IOPL < 3 *)
                    #GP(0); (* Trap to virtual-8086 monitor. *)
                FI;
            FI;
        FI;
    FI;

```

```

Fl;
Fl;
Fl;

```

### Flags Affected

All flags may be affected; see the Operation section for details.

### Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3. If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF EB /r <sup>1</sup>	RM	V/V	MMX	Bitwise OR of <i>mm/m64</i> and <i>mm</i> .
POR <i>mm, mm/m64</i>				
66 OF EB /r	RM	V/V	SSE2	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .
POR <i>xmm1, xmm2/m128</i>				
VEX.NDS.128.66.0F.WIG EB /r	RVM	V/V	AVX	Bitwise OR of <i>xmm2/m128</i> and <i>xmm3</i> .
VPOR <i>xmm1, xmm2, xmm3/m128</i>				
VEX.NDS.256.66.0F.WIG EB /r	RVM	V/V	AVX2	Bitwise OR of <i>ymm2/m256</i> and <i>ymm3</i> .
VPOR <i>ymm1, ymm2, ymm3/m256</i>				

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### POR (128-bit Legacy SSE version)

DEST  $\leftarrow$  DEST OR SRC  
DEST[VLMAX-1:128] (Unmodified)

#### VPOR (VEX.128 encoded version)

DEST  $\leftarrow$  SRC1 OR SRC2  
DEST[VLMAX-1:128]  $\leftarrow$  0

**VPOR (VEX.256 encoded version)**DEST  $\leftarrow$  SRC1 OR SRC2**Intel C/C++ Compiler Intrinsic Equivalent**

POR:       `__m64 _mm_or_si64(__m64 m1, __m64 m2)`  
(V)POR:     `__m128i _mm_or_si128(__m128i m1, __m128i m2)`  
VPOR:      `__m256i _mm256_or_si256 (__m256i a, __m256i b)`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD           If VEX.L = 1.

**PREFETCH<sub>h</sub>—Prefetch Data Into Caches**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 18 /1	PREFETCHT0 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T0 hint.
OF 18 /2	PREFETCHT1 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T1 hint.
OF 18 /3	PREFETCHT2 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T2 hint.
OF 18 /0	PREFETCHNTA <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using NTA hint.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

**Description**

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
  - Pentium III processor—1st- or 2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
  - Pentium III processor—1st-level cache
  - Pentium 4 and Intel Xeon processors—2nd-level cache

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCH<sub>h</sub> instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCH<sub>h</sub> 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<sub>h</sub> instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCH<sub>h</sub> instruction is also

unordered with respect to CLFLUSH instructions, other PREFETCH*h* instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

FETCH (m8);

## Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch(char *p, int i)
```

The argument “\*p” gives the address of the byte (and corresponding cache line) to be prefetched. The value “i” gives a constant (\_MM\_HINT\_T0, \_MM\_HINT\_T1, \_MM\_HINT\_T2, or \_MM\_HINT\_NTA) that specifies the type of prefetch operation to be performed.

## Numeric Exceptions

None.

## Exceptions (All Operating Modes)

#UD                    If the LOCK prefix is used.

## PREFETCHW—Prefetch Data into Caches in Anticipation of a Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF OD /1 PREFETCHW m8	A	V/V	PRFCHW	Move data from m8 closer to the processor in anticipation of a write.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates all other cached instances of the line.

The source operand is a byte memory location. If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates any other cached copy in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH instructions, other PREFETCHW instructions, or any other general instruction.

It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

FETCH\_WITH\_EXCLUSIVE\_OWNERSHIP (m8);

### Flags Affected

All flags are affected

### C/C++ Compiler Intrinsic Equivalent

void \_m\_prefetchw( void \* );

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

#UD                  If the LOCK prefix is used.

**64-Bit Mode Exceptions**

#UD                  If the LOCK prefix is used.

## PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF F6 /r <sup>1</sup> PSADBW mm1, mm2/m64	RM	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from mm2/m64 and mm1; differences are then summed to produce an unsigned word integer result.
66 OF F6 /r PSADBW xmm1, xmm2/m128	RM	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2/m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3/m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from ymm3/m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-10 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

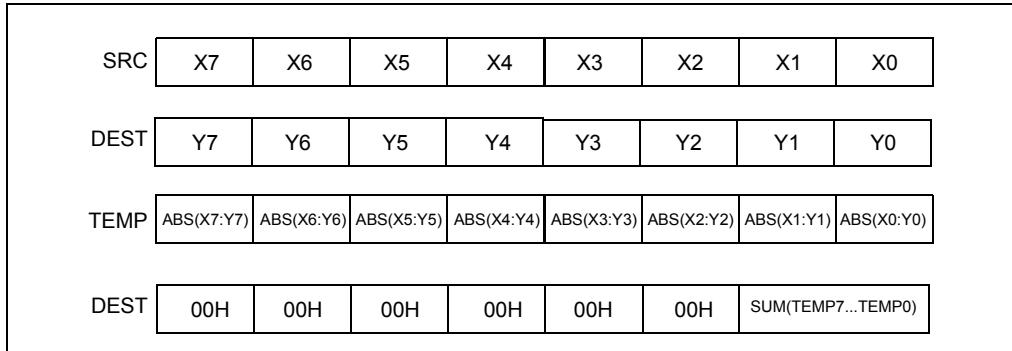


Figure 4-10. PSADBW Instruction Operation Using 64-bit Operands

## Operation

### PSADBW (when using 64-bit operands)

```

TEMPO ← 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(TEMPO:TEMP7);
DEST[63:16] ← 000000000000H;

```

### PSADBW (when using 128-bit operands)

```

TEMPO ← 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(TEMPO:TEMP7);
DEST[63:16] ← 000000000000H;
DEST[79:64] ← SUM(TEMP8:TEMP15);
DEST[127:80] ← 000000000000H;
DEST[VLMAX-1:128] (Unmodified)

```

### VPSADBW (VEX.128 encoded version)

```

TEMPO ← 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(TEMPO:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 000000000000H
DEST[VLMAX-1:128] ← 0

```

### VPSADBW (VEX.256 encoded version)

```

TEMPO ← 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(TEMPO:TEMP7)

```

```

DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[143:128] ← SUM(TEMP16:TEMP23)
DEST[191:144] ← 00000000000H
DEST[207:192] ← SUM(TEMP24:TEMP31)
DEST[223:208] ← 00000000000H

```

### Intel C/C++ Compiler Intrinsic Equivalent

PSADBW:    `__m64 _mm_sad_pu8(__m64 a,__m64 b)`  
 (V)PSADBW: `__m128i _mm_sad_epu8(__m128i a,__m128i b)`  
 VPSADBW: `__m256i _mm256_sad_epu8(__m256i a,__m256i b)`

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally  
 #UD                  If VEX.L = 1.

## PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 00 /r <sup>1</sup> PSHUFB <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Shuffle bytes in <i>mm1</i> according to contents of <i>mm2/m64</i> .
66 OF 38 00 /r PSHUFB <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Shuffle bytes in <i>xmm1</i> according to contents of <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shuffle bytes in <i>xmm2</i> according to contents of <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shuffle bytes in <i>ymm2</i> according to contents of <i>ymm3/m256</i> .

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Legacy SSE version: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask. The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PSHUFB (with 64 bit operands)

```
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
```

```

    DEST[(i*8)+7...(i*8)+0] ← 0;
else
    index[2..0] ← SRC[(i*8)+2 .. (i*8)+0];
    DEST[(i*8)+7...(i*8)+0] ← DEST[(index*8+7)..(index*8+0)];
endif;
}

```

**PSHUFB (with 128 bit operands)**

```

for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
    else
        index[3..0] ← SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← DEST[(index*8+7)..(index*8+0)];
    endif
}
DEST[VLMAX-1:128] ← 0

```

**VPSHUFB (VEX.128 encoded version)**

```

for i = 0 to 15 {
    if (SRC2[(i * 8)+7] = 1) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
    else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[VLMAX-1:128] ← 0

```

**VPSHUFB (VEX.256 encoded version)**

```

for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
    else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
        DEST[128 + (i*8)+7..(i*8)+0] ← 0;
    else
        index[3..0] ← SRC2[128 + (i*8)+3 .. (i*8)+0];
        DEST[128 + (i*8)+7..(i*8)+0] ← SRC1[128 + (index*8+7)..(index*8+0)];
    endif
}

```

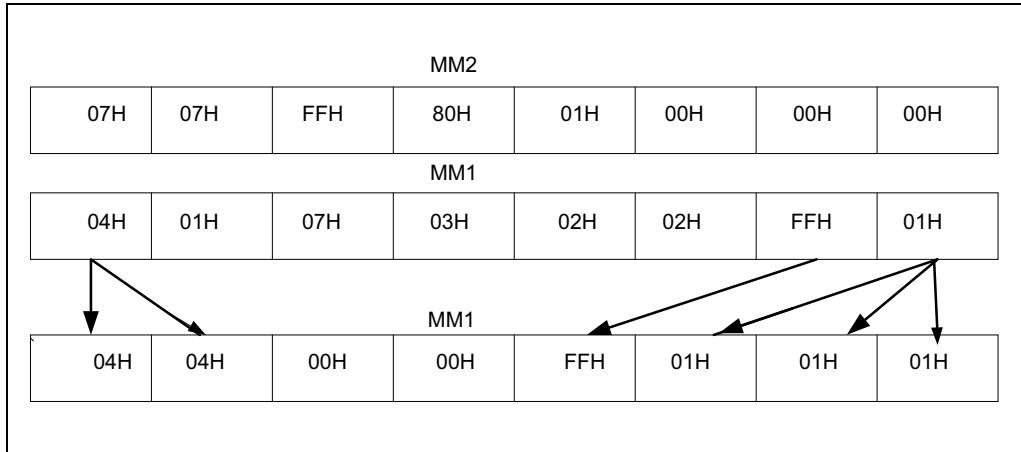


Figure 4-11. PSHUB with 64-Bit Operands

**Intel C/C++ Compiler Intrinsic Equivalent**

PSHUFB:       `__m64 _mm_shuffle_pi8 (__m64 a, __m64 b)`  
 (V)PSHUFB:     `__m128i _mm_shuffle_epi8 (__m128i a, __m128i b)`  
 VPSHUFB:       `__m256i _mm256_shuffle_epi8(__m256i a, __m256i b)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD              If VEX.L = 1.

## PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.66.0F.WIG 70 /r ib VPSHUFD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.66.0F.WIG 70 /r ib VPSHUFD <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the doublewords in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-12 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-12) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

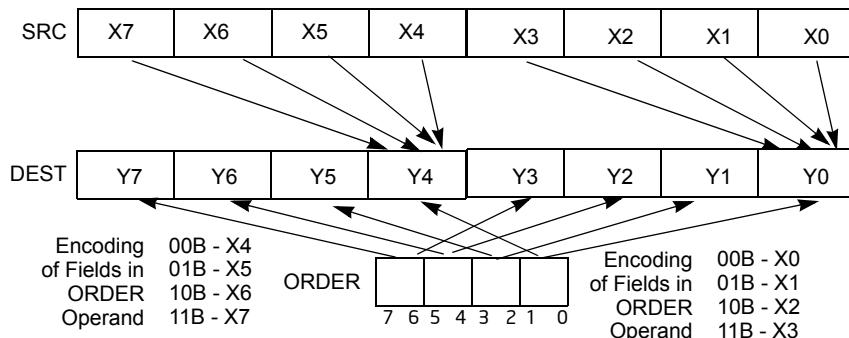


Figure 4-12. 256-bit VPSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

Legacy SSE instructions: In 64-bit mode using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

## Operation

### PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:128] (Unmodified)
```

### VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:128] ← 0
```

### VPSHUFD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
```

## Intel C/C++ Compiler Intrinsic Equivalent

(V)PSHUFD:	<code>__m128i _mm_shuffle_epi32(__m128i a, int n)</code>
VPSHUFD:	<code>__m256i _mm256_shuffle_epi32(__m256i a, const int n)</code>

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

## PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 70 /r ib PSHUFHW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the high words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-12. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

Note: In VEX encoded versions VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### PSHUFHW (128-bit Legacy SSE version)

```

DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] * 16))[79:64]
DEST[95:80] ← (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC >> (imm[7:6] * 16))[79:64]
DEST[VLMAX-1:128] (Unmodified)

```

**VPSHUFHW (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[VLMAX-1:128] ← 0

```

**VPSHUFHW (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] ← SRC1[191:128]
DEST[207:192] ← (SRC1 >> (imm[1:0] *16))[207:192]
DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PSHUFHW: \_\_m128i \_mm\_shufflehi\_epi16(\_\_m128i a, int n)  
 VPSHUFHW: \_\_m256i \_mm256\_shufflehi\_epi16(\_\_m256i a, const int n)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.  
       If VEX.vvvv != 1111B.

## PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 OF 70 /r ib PSHUFLW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the low words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-12. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise instructions will #UD.

### Operation

#### PSHUFLW (128-bit Legacy SSE version)

```

DEST[15:0] ← (SRC >> (imm[1:0] * 16))[15:0]
DEST[31:16] ← (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)

```

**VPSHUFLW (VEX.128 encoded version)**

```

DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[VLMAX-1:128] ← 0

```

**VPSHUFLW (VEX.256 encoded version)**

```

DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC1[127:64]
DEST[143:128] ← (SRC1 >> (imm[1:0] *16))[143:128]
DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
DEST[255:192] ← SRC1[255:192]

```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PSHUFLW:    \_\_m128i \_mm\_shufflelo\_epi16(\_\_m128i a, int n)  
 VPSHUFLW:    \_\_m256i \_mm256\_shufflelo\_epi16(\_\_m256i a, const int n)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD               If VEX.L = 1.  
                If VEX.vvvv != 1111B.

**PSHUFW—Shuffle Packed Words**

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 70 /r ib PSHUFW mm1, mm2/m64, imm8	RMI	Valid	Valid	Shuffle the words in <i>mm2/m64</i> based on the encoding in <i>imm8</i> and store the result in <i>mm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

**Description**

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-12. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

```
DEST[15:0] ← (SRC >> (ORDER[1:0] * 16))[15:0];
DEST[31:16] ← (SRC >> (ORDER[3:2] * 16))[15:0];
DEST[47:32] ← (SRC >> (ORDER[5:4] * 16))[15:0];
DEST[63:48] ← (SRC >> (ORDER[7:6] * 16))[15:0];
```

**Intel C/C++ Compiler Intrinsic Equivalent**

PSHUFW:     \_\_m64 \_mm\_shuffle\_pi16(\_\_m64 a, int n)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Table 22-7, "Exception Conditions for SIMD/MMX Instructions with Memory Reference," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

## PSIGNB/PSIGNW/PSIGND — Packed SIGN

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 08 /r <sup>1</sup> PSIGNB <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m64</i> .
66 0F 38 08 /r PSIGNB <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
0F 38 09 /r <sup>1</sup> PSIGNW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m128</i> .
66 0F 38 09 /r PSIGNW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
0F 38 0A /r <sup>1</sup> PSIGND <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m128</i> .
66 0F 38 0A /r PSIGND <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 08 /r VPSIGNB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed byte integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.128.66.0F38.WIG 09 /r VPSIGNW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed word integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.128.66.0F38.WIG 0A /r VPSIGND <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed doubleword integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 08 /r VPSIGNB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Negate packed byte integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.
VEX.NDS.256.66.0F38.WIG 09 /r VPSIGNW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Negate packed 16-bit integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.
VEX.NDS.256.66.0F38.WIG 0A /r VPSIGND <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Negate packed doubleword integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.

### NOTES:

- See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers. When the source operand is a 128bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

## Operation

### PSIGNB (with 64 bit operands)

```

IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 7th bytes

```

```

IF (SRC[63:56] < 0 )
    DEST[63:56] ← Neg(DEST[63:56])
ELSEIF (SRC[63:56] = 0 )
    DEST[63:56] ← 0
ELSEIF (SRC[63:56] > 0 )
    DEST[63:56] ← DEST[63:56]

```

### PSIGNB (with 128 bit operands)

```

IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 15th bytes
IF (SRC[127:120] < 0 )
    DEST[127:120] ← Neg(DEST[127:120])
ELSEIF (SRC[127:120] = 0 )
    DEST[127:120] ← 0
ELSEIF (SRC[127:120] > 0 )
    DEST[127:120] ← DEST[127:120]

```

**VPSIGNB (VEX.128 encoded version)**

```
DEST[127:0] ← BYTE_SIGN(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

**VPSIGNB (VEX.256 encoded version)**

```
DEST[255:0] ← BYTE_SIGN_256b(SRC1, SRC2)
```

**PSIGNW (with 64 bit operands)**

```
IF (SRC[15:0] < 0 )
    DEST[15:0] ← Neg(DEST[15:0])
ELSEIF (SRC[15:0] = 0 )
    DEST[15:0] ← 0
ELSEIF (SRC[15:0] > 0 )
    DEST[15:0] ← DEST[15:0]
```

Repeat operation for 2nd through 3rd words

```
IF (SRC[63:48] < 0 )
    DEST[63:48] ← Neg(DEST[63:48])
ELSEIF (SRC[63:48] = 0 )
    DEST[63:48] ← 0
ELSEIF (SRC[63:48] > 0 )
    DEST[63:48] ← DEST[63:48]
```

**PSIGNW (with 128 bit operands)**

```
IF (SRC[15:0] < 0 )
    DEST[15:0] ← Neg(DEST[15:0])
ELSEIF (SRC[15:0] = 0 )
    DEST[15:0] ← 0
ELSEIF (SRC[15:0] > 0 )
    DEST[15:0] ← DEST[15:0]
```

Repeat operation for 2nd through 7th words

```
IF (SRC[127:112] < 0 )
    DEST[127:112] ← Neg(DEST[127:112])
ELSEIF (SRC[127:112] = 0 )
    DEST[127:112] ← 0
ELSEIF (SRC[127:112] > 0 )
    DEST[127:112] ← DEST[127:112]
```

**VPSIGNW (VEX.128 encoded version)**

```
DEST[127:0] ← WORD_SIGN(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

**VPSIGNW (VEX.256 encoded version)**

```
DEST[255:0] ← WORD_SIGN(SRC1, SRC2)
```

**PSIGND (with 64 bit operands)**

```
IF (SRC[31:0] < 0 )
    DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
    DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
    DEST[31:0] ← DEST[31:0]
IF (SRC[63:32] < 0 )
    DEST[63:32] ← Neg(DEST[63:32])
ELSEIF (SRC[63:32] = 0 )
    DEST[63:32] ← 0
```

```
ELSEIF (SRC[63:32] > 0 )
    DEST[63:32] ← DEST[63:32]
```

**PSIGND (with 128 bit operands)**

```
IF (SRC[31:0] < 0 )
    DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
    DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
    DEST[31:0] ← DEST[31:0]
Repeat operation for 2nd through 3rd double words
IF (SRC[127:96] < 0 )
    DEST[127:96] ← Neg(DEST[127:96])
ELSEIF (SRC[127:96] = 0 )
    DEST[127:96] ← 0
ELSEIF (SRC[127:96] > 0 )
    DEST[127:96] ← DEST[127:96]
```

**VPSIGND (VEX.128 encoded version)**

```
DEST[127:0] ← DWORD_SIGN(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

**VPSIGND (VEX.256 encoded version)**

```
DEST[255:0] ← DWORD_SIGN(SRC1, SRC2)
```

**Intel C/C++ Compiler Intrinsic Equivalent**

PSIGNB:	<code>__m64 _mm_sign_pi8 (__m64 a, __m64 b)</code>
(V)PSIGNB:	<code>__m128i _mm_sign_epi8 (__m128i a, __m128i b)</code>
VPSIGNB:	<code>__m256i _mm256_sign_epi8 (__m256i a, __m256i b)</code>
PSIGNW:	<code>__m64 _mm_sign_pi16 (__m64 a, __m64 b)</code>
(V)PSIGNW:	<code>__m128i _mm_sign_epi16 (__m128i a, __m128i b)</code>
VPSIGNW:	<code>__m256i _mm256_sign_epi16 (__m256i a, __m256i b)</code>
PSIGND:	<code>__m64 _mm_sign_pi32 (__m64 a, __m64 b)</code>
(V)PSIGND:	<code>__m128i _mm_sign_epi32 (__m128i a, __m128i b)</code>
VPSIGND:	<code>__m256i _mm256_sign_epi32 (__m256i a, __m256i b)</code>

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift <i>xmm1</i> left by <i>imm8</i> bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift <i>xmm2</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>xmm1</i> .
VEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ <i>ymm1, ymm2, imm8</i>	VMI	V/V	AVX2	Shift <i>ymm2</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	imm8	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

### Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

Note: VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

### Operation

#### PSLLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST << (TEMP * 8)
DEST[VLMAX-1:128] ← 0
```

#### VPSLLDQ (VEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC << (TEMP * 8)
DEST[VLMAX-1:128] ← 0
```

#### VPSLLDQ (VEX.256 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] << (TEMP * 8)
DEST[255:128] ← SRC[255:128] << (TEMP * 8)
```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PSLLDQ: `__m128i _mm_slli_si128 ( __m128i a, int imm)`  
VPSLLDQ: `__m256i _mm256_slli_si256 ( __m256i a, const int imm)`

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 7; additionally

#UD                    If VEX.L = 1.

## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F1 /r <sup>1</sup> PSLLW <i>mm, mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> left <i>mm/m64</i> while shifting in 0s.
66 0F F1 /r PSLLW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 71 /6 ib PSLLW <i>mm1, imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 71 /6 ib PSLLW <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F2 /r <sup>1</sup> PSLLD <i>mm, mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F2 /r PSLLD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Shift doublewords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 72 /6 ib <sup>1</sup> PSLLD <i>mm, imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 72 /6 ib PSLLD <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F3 /r <sup>1</sup> PSLLQ <i>mm, mm/m64</i>	RM	V/V	MMX	Shift quadword in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F3 /r PSLLQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Shift quadwords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 73 /6 ib <sup>1</sup> PSLLQ <i>mm, imm8</i>	MI	V/V	MMX	Shift quadword in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 73 /6 ib PSLLQ <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift quadwords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shift quadwords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift quadwords in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG F1 /r VPSLLW <i>ymm1, ymm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW <i>ymm1, ymm2, imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.

VEX.NDS.256.66.0F.WIG F2 /r VPSLLD <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift doublewords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift doublewords in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift quadwords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift quadwords in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.

**NOTES:**

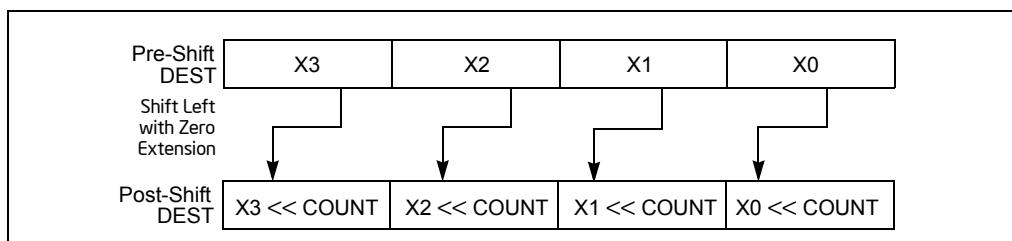
1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-13 gives an example of shifting words in a 64-bit operand.



**Figure 4-13. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand**

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 8-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

## Operation

### PSLLW (with 64-bit operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
FI;
```

### PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
FI;
```

### PSLLQ (with 64-bit operand)

```
IF (COUNT > 63)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST ← ZeroExtend(DEST << COUNT);
FI;
```

### PSLLW (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H;
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(DEST[127:112] << COUNT);
FI;
```

### PSLLD (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
```

(\* Repeat shift operation for 2nd and 3rd doublewords \*)  
 DEST[127:96] ← ZeroExtend(DEST[127:96] << COUNT);  
 FI;

**PSLLQ (with 128-bit operand)**

COUNT ← COUNT\_SOURCE[63:0];  
 IF (COUNT > 63)  
 THEN  
   DEST[128:0] ← 00000000000000000000000000000000H;  
 ELSE  
   DEST[63:0] ← ZeroExtend(DEST[63:0] << COUNT);  
   DEST[127:64] ← ZeroExtend(DEST[127:64] << COUNT);  
 FI;

**PSLLW (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, SRC)  
 DEST[VLMAX-1:128] (Unmodified)

**PSLLW (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, imm8)  
 DEST[VLMAX-1:128] (Unmodified)

**VPSLLD (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(SRC1, SRC2)  
 DEST[VLMAX-1:128] ← 0

**VPSLLD (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(SRC1, imm8)  
 DEST[VLMAX-1:128] ← 0

**PSLLD (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(DEST, SRC)  
 DEST[VLMAX-1:128] (Unmodified)

**PSLLD (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(DEST, imm8)  
 DEST[VLMAX-1:128] (Unmodified)

**VPSLLQ (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS(SRC1, SRC2)  
 DEST[VLMAX-1:128] ← 0

**VPSLLQ (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS(SRC1, imm8)  
 DEST[VLMAX-1:128] ← 0

**PSLLQ (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS(DEST, SRC)  
 DEST[VLMAX-1:128] (Unmodified)

**PSLLQ (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS(DEST, imm8)  
 DEST[VLMAX-1:128] (Unmodified)

**VPSLLW (xmm, xmm, xmm/m128)**

DEST[127:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_WORDS(SRC1, SRC2)  
 DEST[VLMAX-1:128]  $\leftarrow$  0

**VPSLLW (xmm, imm8)**

DEST[127:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_WORDS(SRC1, imm8)  
 DEST[VLMAX-1:128]  $\leftarrow$  0

**PSLLW (xmm, xmm, xmm/m128)**

DEST[127:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, SRC)  
 DEST[VLMAX-1:128] (Unmodified)

**PSLLW (xmm, imm8)**

DEST[127:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, imm8)  
 DEST[VLMAX-1:128] (Unmodified)

**VPSLLD (xmm, xmm, xmm/m128)**

DEST[127:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_DWORDS(SRC1, SRC2)  
 DEST[VLMAX-1:128]  $\leftarrow$  0

**VPSLLD (xmm, imm8)**

DEST[127:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_DWORDS(SRC1, imm8)  
 DEST[VLMAX-1:128]  $\leftarrow$  0

**VPSLLW (ymm, ymm, xmm/m128)**

DEST[255:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

**VPSLLW (ymm, imm8)**

DEST[255:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_WORD\_256b(SRC1, imm8)

**VPSLLD (ymm, ymm, xmm/m128)**

DEST[255:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1, SRC2)

**VPSLLD (ymm, imm8)**

DEST[127:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1, imm8)

**VPSLLQ (ymm, ymm, xmm/m128)**

DEST[255:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1, SRC2)

**VPSLLQ (ymm, imm8)**

DEST[255:0]  $\leftarrow$  LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1, imm8)

**Intel C/C++ Compiler Intrinsic Equivalents**

PSLLW:	<code>__m64 _mm_slli_pi16 (__m64 m, int count)</code>
PSLLW:	<code>__m64 _mm_sll_pi16(__m64 m, __m64 count)</code>
(V)PSLLW:	<code>__m128i _mm_slli_pi16(__m64 m, int count)</code>
(V)PSLLW:	<code>__m128i _mm_slli_pi16(__m128i m, __m128i count)</code>
VPSLLW:	<code>__m256i _mm256_slli_epi16 (__m256i m, int count)</code>
VPSLLW:	<code>__m256i _mm256_sll_epi16 (__m256i m, __m128i count)</code>
PSLLD:	<code>__m64 _mm_slli_pi32(__m64 m, int count)</code>
PSLLD:	<code>__m64 _mm_sll_pi32(__m64 m, __m64 count)</code>

(V)PSLLD:  $\text{__m128i}_\text{mm\_slli\_epi32}(\text{__m128i } m, \text{int } count)$   
 (V)PSLLD:  $\text{__m128i}_\text{mm\_sll\_epi32}(\text{__m128i } m, \text{__m128i } count)$   
 VPSLLD:  $\text{__m256i}_\text{mm256\_slli\_epi32}(\text{__m256i } m, \text{int } count)$   
 VPSLLD:  $\text{__m256i}_\text{mm256\_sll\_epi32}(\text{__m256i } m, \text{__m128i } count)$   
 PSLLQ:  $\text{__m64}_\text{mm\_slli\_si64}(\text{__m64 } m, \text{int } count)$   
 PSLLQ:  $\text{__m64}_\text{mm\_sll\_si64}(\text{__m64 } m, \text{__m64 } count)$   
 (V)PSLLQ:  $\text{__m128i}_\text{mm\_slli\_epi64}(\text{__m128i } m, \text{int } count)$   
 (V)PSLLQ:  $\text{__m128i}_\text{mm\_sll\_epi64}(\text{__m128i } m, \text{__m128i } count)$   
 VPSLLQ:  $\text{__m256i}_\text{mm256\_slli\_epi64}(\text{__m256i } m, \text{int } count)$   
 VPSLLQ:  $\text{__m256i}_\text{mm256\_sll\_epi64}(\text{__m256i } m, \text{__m128i } count)$

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally

#UD                    If VEX.L = 1.

## PSRAW/PSRAD—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF E1 / <sup>1</sup> PSRAW <i>mm, mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 OF E1 / <i>r</i> PSRAW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
OF 71 /4 ib <sup>1</sup> PSRAW <i>mm, imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in sign bits
66 OF 71 /4 ib PSRAW <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits
OF E2 / <sup>1</sup> PSRAD <i>mm, mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 OF E2 / <i>r</i> PSRAD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Shift doubleword in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
OF 72 /4 ib <sup>1</sup> PSRAD <i>mm, imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 OF 72 /4 ib PSRAD <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 / <i>r</i> VPSRAW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 / <i>r</i> VPSRAD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E1 / <i>r</i> VPSRAW <i>ymm1, ymm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW <i>ymm1, ymm2, imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E2 / <i>r</i> VPSRAD <i>ymm1, ymm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 72 /4 ib VPSRAD <i>ymm1, ymm2, imm8</i>	VMI	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by <i>imm8</i> while shifting in sign bits.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding				
Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

### Description

Shifts the bits in the individual data elements (words or doublewords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words) or 31 (for doublewords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-14 gives an example of shifting words in a 64-bit operand.)

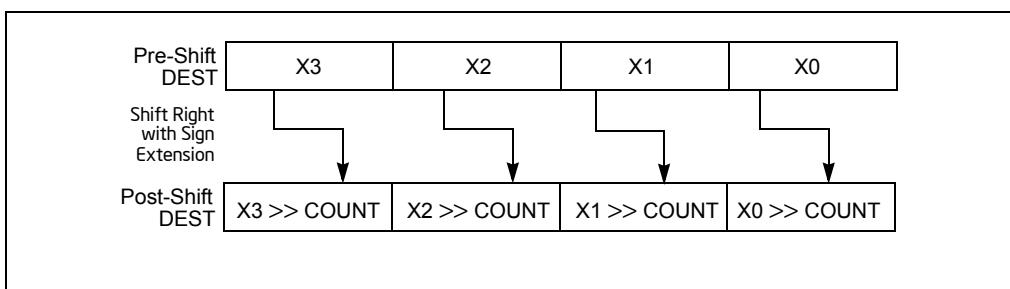


Figure 4-14. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE instructions:** The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 8-bit immediate.

**128-bit Legacy SSE version:** The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

**VEX.128 encoded version:** The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

**VEX.256 encoded version:** The destination and first source operands are YMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

**Note:** For shifts with an immediate count (VEX.128.66.0F 71-73 /4), VEX.vvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

## Operation

### PSRAW (with 64-bit operand)

```

IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] ← SignExtend(DEST[63:48] >> COUNT);

```

### PSRAD (with 64-bit operand)

```

IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
DEST[63:32] ← SignExtend(DEST[63:32] >> COUNT);

```

### PSRAW (with 128-bit operand)

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(DEST[127:112] >> COUNT);

```

### PSRAD (with 128-bit operand)

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd doublewords *)
DEST[127:96] ← SignExtend(DEST[127:96] >> COUNT);

```

### PSRAW (xmm, xmm, xmm/m128)

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

```

### PSRAW (xmm, imm8)

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

```

### VPSRAW (xmm, xmm, xmm/m128)

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

### VPSRAW (xmm, imm8)

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

```

**PSRAD (xmm, xmm, xmm/m128)**

`DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORD(SRC, DEST)`  
`DEST[VLMAX-1:128] (Unmodified)`

**PSRAD (xmm, imm8)**

`DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORD(imm8, DEST)`  
`DEST[VLMAX-1:128] (Unmodified)`

**VPSRAD (xmm, xmm, xmm/m128)**

`DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORD(SRC1, SRC2)`  
`DEST[VLMAX-1:128] ← 0`

**VPSRAD (xmm, imm8)**

`DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORD(SRC1, imm8)`  
`DEST[VLMAX-1:128] ← 0`

**VPSRAW (ymm, ymm, xmm/m128)**

`DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)`

**VPSRAW (ymm, imm8)**

`DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)`

**VPSRAD (ymm, ymm, xmm/m128)**

`DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORD_256b(SRC1, SRC2)`

**VPSRAD (ymm, imm8)**

`DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORD_256b(SRC1, imm8)`

**Intel C/C++ Compiler Intrinsic Equivalents**

PSRAW:	<code>_m64 _mm_srai_pi16 (_m64 m, int count)</code>
PSRAW:	<code>_m64 _mm_sra_pi16 (_m64 m, _m64 count)</code>
(V)PSRAW:	<code>_m128i _mm_srai_epi16(_m128i m, int count)</code>
(V)PSRAW:	<code>_m128i _mm_sra_epi16(_m128i m, _m128i count)</code>
VPSRAW:	<code>_m256i _mm256_srai_epi16 (_m256i m, int count)</code>
VPSRAW:	<code>_m256i _mm256_sra_epi16 (_m256i m, _m128i count)</code>
PSRAD:	<code>_m64 _mm_srai_pi32 (_m64 m, int count)</code>
PSRAD:	<code>_m64 _mm_sra_pi32 (_m64 m, _m64 count)</code>
(V)PSRAD:	<code>_m128i _mm_srai_epi32 (_m128i m, int count)</code>
(V)PSRAD:	<code>_m128i _mm_sra_epi32 (_m128i m, _m128i count)</code>
VPSRAD:	<code>_m256i _mm256_srai_epi32 (_m256i m, int count)</code>
VPSRAD:	<code>_m256i _mm256_sra_epi32 (_m256i m, _m128i count)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

### Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally  
#UD                    If VEX.L = 1.

## PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift <i>xmm2</i> right by <i>imm8</i> bytes while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ <i>ymm1, ymm2, imm8</i>	VMI	V/V	AVX2	Shift <i>ymm1</i> right by <i>imm8</i> bytes while shifting in 0s.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	imm8	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

### Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

Note: VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

### Operation

#### PSRLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST >> (TEMP * 8)
DEST[VLMAX-1:128] ← 0 (Unmodified)
```

#### VPSRLDQ (VEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC >> (TEMP * 8)
DEST[VLMAX-1:128] ← 0
```

#### VPSRLDQ (VEX.256 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] >> (TEMP * 8)
DEST[255:128] ← SRC[255:128] >> (TEMP * 8)
```

## Intel C/C++ Compiler Intrinsic Equivalents

(V)PSRLDQ: `__m128i_mm_srli_si128 ( __m128i a, int imm)`  
VPSRLDQ: `__m256i_mm256_srli_si256 ( __m256i a, const int imm)`

## Flags Affected

None.

## Numeric Exceptions

None.

## Other Exceptions

See Exceptions Type 7; additionally

#UD                  If VEX.L = 1.

## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF D1 /r <sup>1</sup> PSRLW <i>mm, mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 OF D1 /r PSRLW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
OF 71 /2 ib <sup>1</sup> PSRLW <i>mm, imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 OF 71 /2 ib PSRLW <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
OF D2 /r <sup>1</sup> PSRLD <i>mm, mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 OF D2 /r PSRLD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
OF 72 /2 ib <sup>1</sup> PSRLD <i>mm, imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 OF 72 /2 ib PSRLD <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
OF D3 /r <sup>1</sup> PSRLQ <i>mm, mm/m64</i>	RM	V/V	MMX	Shift <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 OF D3 /r PSRLQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Shift quadwords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
OF 73 /2 ib <sup>1</sup> PSRLQ <i>mm, imm8</i>	MI	V/V	MMX	Shift <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 OF 73 /2 ib PSRLQ <i>xmm1, imm8</i>	MI	V/V	SSE2	Shift quadwords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shift quadwords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ <i>xmm1, xmm2, imm8</i>	VMI	V/V	AVX	Shift quadwords in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW <i>ymm1, ymm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW <i>ymm1, ymm2, imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.

VEX.NDS.256.66.0F.WIG D2 /r VPSRLD <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift doublewords in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift quadwords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift quadwords in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

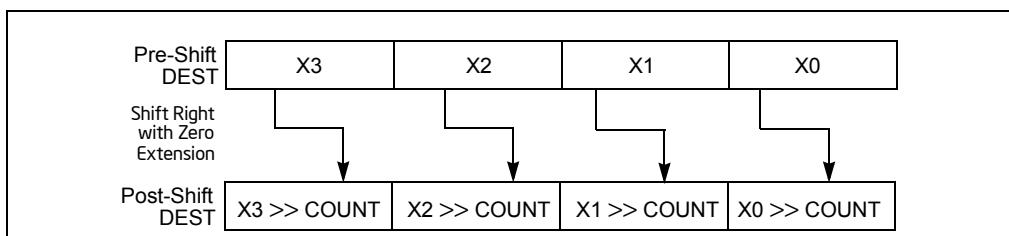
**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-15 gives an example of shifting words in a 64-bit operand.

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.



**Figure 4-15. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand**

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE instructions:** The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 8-bit immediate.

**128-bit Legacy SSE version:** The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits

are loaded but the upper 64 bits are ignored. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 128-bit memory location or an 8-bit immediate.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

## Operation

### PSRLW (with 64-bit operand)

```

IF (COUNT > 15)
THEN
    DEST[64:0] ← 0000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
FI;
```

### PSRLD (with 64-bit operand)

```

IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
FI;
```

### PSRLQ (with 64-bit operand)

```

IF (COUNT > 63)
THEN
    DEST[64:0] ← 0000000000000000H
ELSE
    DEST ← ZeroExtend(DEST >> COUNT);
FI;
```

### PSRLW (with 128-bit operand)

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(DEST[127:112] >> COUNT);
FI;
```

### PSRLD (with 128-bit operand)

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H
```

```

ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd doublewords *)
    DEST[127:96] ← ZeroExtend(DEST[127:96] >> COUNT);
FI;

```

**PSRLQ (with 128-bit operand)**

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(DEST[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(DEST[127:64] >> COUNT);
FI;

```

**PSRLW (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

```

**PSRLW (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

```

**VPSRLW (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

**VPSRLW (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

```

**PSRLD (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

```

**PSRLD (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

```

**VPSRLD (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

**VPSRLD (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

```

**PSRLQ (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

```

**PSRLQ (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)

```

DEST[VLMAX-1:128] (Unmodified)

**VPSRLQ (xmm, xmm, xmm/m128)**

DEST[127:0]  $\leftarrow$  LOGICAL\_RIGHT\_SHIFT\_QWORDS(SRC1, SRC2)

DEST[VLMAX-1:128]  $\leftarrow$  0

**VPSRLQ (xmm, imm8)**

DEST[127:0]  $\leftarrow$  LOGICAL\_RIGHT\_SHIFT\_QWORDS(SRC1, imm8)

DEST[VLMAX-1:128]  $\leftarrow$  0

**VPSRLW (ymm, ymm, xmm/m128)**

DEST[255:0]  $\leftarrow$  LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

**VPSRLW (ymm, imm8)**

DEST[255:0]  $\leftarrow$  LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, imm8)

**VPSRLD (ymm, ymm, xmm/m128)**

DEST[255:0]  $\leftarrow$  LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1, SRC2)

**VPSRLD (ymm, imm8)**

DEST[255:0]  $\leftarrow$  LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1, imm8)

**VPSRLQ (ymm, ymm, xmm/m128)**

DEST[255:0]  $\leftarrow$  LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1, SRC2)

**VPSRLQ (ymm, imm8)**

DEST[255:0]  $\leftarrow$  LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1, imm8)

### Intel C/C++ Compiler Intrinsic Equivalents

PSRLW:	<code>_m64 _mm_srli_pi16(_m64 m, int count)</code>
PSRLW:	<code>_m64 _mm_srl_pi16 (_m64 m, __m64 count)</code>
(V)PSRLW:	<code>_m128i _mm_srli_epi16 (_m128i m, int count)</code>
(V)PSRLW:	<code>_m128i _mm_srl_epi16 (_m128i m, __m128i count)</code>
VPSRLW:	<code>_m256i _mm256_srli_epi16 (_m256i m, int count)</code>
VPSRLW:	<code>_m256i _mm256_srl_epi16 (_m256i m, __m128i count)</code>
PSRLD:	<code>_m64 _mm_srli_pi32 (_m64 m, int count)</code>
PSRDL:	<code>_m64 _mm_srl_pi32 (_m64 m, __m64 count)</code>
(V)PSRDL:	<code>_m128i _mm_srli_epi32 (_m128i m, int count)</code>
(V)PSRDL:	<code>_m128i _mm_srl_epi32 (_m128i m, __m128i count)</code>
VPSRDL:	<code>_m256i _mm256_srli_epi32 (_m256i m, int count)</code>
VPSRDL:	<code>_m256i _mm256_srl_epi32 (_m256i m, __m128i count)</code>
PSRLQ:	<code>_m64 _mm_srli_si64 (_m64 m, int count)</code>
PSRLQ:	<code>_m64 _mm_srl_si64 (_m64 m, __m64 count)</code>
(V)PSRLQ:	<code>_m128i _mm_srli_epi64 (_m128i m, int count)</code>
(V)PSRLQ:	<code>_m128i _mm_srl_epi64 (_m128i m, __m128i count)</code>
VPSRLQ:	<code>_m256i _mm256_srli_epi64 (_m256i m, int count)</code>
VPSRLQ:	<code>_m256i _mm256_srl_epi64 (_m256i m, __m128i count)</code>

## Flags Affected

None.

## Numeric Exceptions

None.

## Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally

#UD                    If VEX.L = 1.

## PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F8 / <sup>1</sup> PSUBB <i>mm, mm/m64</i>	RM	V/V	MMX	Subtract packed byte integers in <i>mm/m64</i> from packed byte integers in <i>mm</i> .
66 0F F8 / <i>r</i> PSUBB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Subtract packed byte integers in <i>xmm2/m128</i> from packed byte integers in <i>xmm1</i> .
0F F9 / <sup>1</sup> PSUBW <i>mm, mm/m64</i>	RM	V/V	MMX	Subtract packed word integers in <i>mm/m64</i> from packed word integers in <i>mm</i> .
66 0F F9 / <i>r</i> PSUBW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Subtract packed word integers in <i>xmm2/m128</i> from packed word integers in <i>xmm1</i> .
OF FA / <sup>1</sup> PSUBD <i>mm, mm/m64</i>	RM	V/V	MMX	Subtract packed doubleword integers in <i>mm/m64</i> from packed doubleword integers in <i>mm</i> .
66 OF FA / <i>r</i> PSUBD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Subtract packed doubleword integers in <i>xmm2/m128</i> from packed doubleword integers in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F8 / <i>r</i> VPSUBB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Subtract packed byte integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG F9 / <i>r</i> VPSUBW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Subtract packed word integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG FA / <i>r</i> VPSUBD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Subtract packed doubleword integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG F8 / <i>r</i> VPSUBB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed byte integers in <i>ymm3/m256</i> from <i>ymm2</i> .
VEX.NDS.256.66.0F.WIG F9 / <i>r</i> VPSUBW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed word integers in <i>ymm3/m256</i> from <i>ymm2</i> .
VEX.NDS.256.66.0F.WIG FA / <i>r</i> VPSUBD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed doubleword integers in <i>ymm3/m256</i> from <i>ymm2</i> .

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE version:** When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

**128-bit Legacy SSE version:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded version:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**VEX.256 encoded version:** The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

## Operation

### PSUBB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] – SRC[7:0];
(* Repeat subtract operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] – SRC[63:56];
```

### PSUBB (with 128-bit operands)

```
DEST[7:0] ← DEST[7:0] – SRC[7:0];
(* Repeat subtract operation for 2nd through 14th byte *)
DEST[127:120] ← DEST[111:120] – SRC[127:120];
```

### VPSUBB (VEX.128 encoded version)

```
DEST[7:0] ← SRC1[7:0]-SRC2[7:0]
DEST[15:8] ← SRC1[15:8]-SRC2[15:8]
DEST[23:16] ← SRC1[23:16]-SRC2[23:16]
DEST[31:24] ← SRC1[31:24]-SRC2[31:24]
DEST[39:32] ← SRC1[39:32]-SRC2[39:32]
DEST[47:40] ← SRC1[47:40]-SRC2[47:40]
DEST[55:48] ← SRC1[55:48]-SRC2[55:48]
DEST[63:56] ← SRC1[63:56]-SRC2[63:56]
DEST[71:64] ← SRC1[71:64]-SRC2[71:64]
DEST[79:72] ← SRC1[79:72]-SRC2[79:72]
DEST[87:80] ← SRC1[87:80]-SRC2[87:80]
DEST[95:88] ← SRC1[95:88]-SRC2[95:88]
DEST[103:96] ← SRC1[103:96]-SRC2[103:96]
DEST[111:104] ← SRC1[111:104]-SRC2[111:104]
DEST[119:112] ← SRC1[119:112]-SRC2[119:112]
DEST[127:120] ← SRC1[127:120]-SRC2[127:120]
DEST[VLMAX-1:128] ← 00
```

**VPSUBB (VEX.256 encoded version)**

DEST[7:0]  $\leftarrow$  SRC1[7:0]-SRC2[7:0]  
 DEST[15:8]  $\leftarrow$  SRC1[15:8]-SRC2[15:8]  
 DEST[23:16]  $\leftarrow$  SRC1[23:16]-SRC2[23:16]  
 DEST[31:24]  $\leftarrow$  SRC1[31:24]-SRC2[31:24]  
 DEST[39:32]  $\leftarrow$  SRC1[39:32]-SRC2[39:32]  
 DEST[47:40]  $\leftarrow$  SRC1[47:40]-SRC2[47:40]  
 DEST[55:48]  $\leftarrow$  SRC1[55:48]-SRC2[55:48]  
 DEST[63:56]  $\leftarrow$  SRC1[63:56]-SRC2[63:56]  
 DEST[71:64]  $\leftarrow$  SRC1[71:64]-SRC2[71:64]  
 DEST[79:72]  $\leftarrow$  SRC1[79:72]-SRC2[79:72]  
 DEST[87:80]  $\leftarrow$  SRC1[87:80]-SRC2[87:80]  
 DEST[95:88]  $\leftarrow$  SRC1[95:88]-SRC2[95:88]  
 DEST[103:96]  $\leftarrow$  SRC1[103:96]-SRC2[103:96]  
 DEST[111:104]  $\leftarrow$  SRC1[111:104]-SRC2[111:104]  
 DEST[119:112]  $\leftarrow$  SRC1[119:112]-SRC2[119:112]  
 DEST[127:120]  $\leftarrow$  SRC1[127:120]-SRC2[127:120]  
 DEST[135:128]  $\leftarrow$  SRC1[135:128]-SRC2[135:128]  
 DEST[143:136]  $\leftarrow$  SRC1[143:136]-SRC2[143:136]  
 DEST[151:144]  $\leftarrow$  SRC1[151:144]-SRC2[151:144]  
 DEST[159:152]  $\leftarrow$  SRC1[159:152]-SRC2[159:152]  
 DEST[167:160]  $\leftarrow$  SRC1[167:160]-SRC2[167:160]  
 DEST[175:168]  $\leftarrow$  SRC1[175:168]-SRC2[175:168]  
 DEST[183:176]  $\leftarrow$  SRC1[183:176]-SRC2[183:176]  
 DEST[191:184]  $\leftarrow$  SRC1[191:184]-SRC2[191:184]  
 DEST[199:192]  $\leftarrow$  SRC1[199:192]-SRC2[199:192]  
 DEST[207:200]  $\leftarrow$  SRC1[207:200]-SRC2[207:200]  
 DEST[215:208]  $\leftarrow$  SRC1[215:208]-SRC2[215:208]  
 DEST[223:216]  $\leftarrow$  SRC1[223:216]-SRC2[223:216]  
 DEST[231:224]  $\leftarrow$  SRC1[231:224]-SRC2[231:224]  
 DEST[239:232]  $\leftarrow$  SRC1[239:232]-SRC2[239:232]  
 DEST[247:240]  $\leftarrow$  SRC1[247:240]-SRC2[247:240]  
 DEST[255:248]  $\leftarrow$  SRC1[255:248]-SRC2[255:248]

**PSUBW (with 64-bit operands)**

DEST[15:0]  $\leftarrow$  DEST[15:0] – SRC[15:0];  
 (\* Repeat subtract operation for 2nd and 3rd word \*)  
 DEST[63:48]  $\leftarrow$  DEST[63:48] – SRC[63:48];

**PSUBW (with 128-bit operands)**

DEST[15:0]  $\leftarrow$  DEST[15:0] – SRC[15:0];  
 (\* Repeat subtract operation for 2nd through 7th word \*)  
 DEST[127:112]  $\leftarrow$  DEST[127:112] – SRC[127:112];

**VPSUBW (VEX.128 encoded version)**

DEST[15:0]  $\leftarrow$  SRC1[15:0]-SRC2[15:0]  
 DEST[31:16]  $\leftarrow$  SRC1[31:16]-SRC2[31:16]  
 DEST[47:32]  $\leftarrow$  SRC1[47:32]-SRC2[47:32]  
 DEST[63:48]  $\leftarrow$  SRC1[63:48]-SRC2[63:48]  
 DEST[79:64]  $\leftarrow$  SRC1[79:64]-SRC2[79:64]  
 DEST[95:80]  $\leftarrow$  SRC1[95:80]-SRC2[95:80]  
 DEST[111:96]  $\leftarrow$  SRC1[111:96]-SRC2[111:96]  
 DEST[127:112]  $\leftarrow$  SRC1[127:112]-SRC2[127:112]

`DEST[VLMAX-1:128] ← 0`

#### VPSUBW (VEX.256 encoded version)

```

DEST[15:0] ← SRC1[15:0]-SRC2[15:0]
DEST[31:16] ← SRC1[31:16]-SRC2[31:16]
DEST[47:32] ← SRC1[47:32]-SRC2[47:32]
DEST[63:48] ← SRC1[63:48]-SRC2[63:48]
DEST[79:64] ← SRC1[79:64]-SRC2[79:64]
DEST[95:80] ← SRC1[95:80]-SRC2[95:80]
DEST[111:96] ← SRC1[111:96]-SRC2[111:96]
DEST[127:112] ← SRC1[127:112]-SRC2[127:112]
DEST[143:128] ← SRC1[143:128]-SRC2[143:128]
DEST[159:144] ← SRC1[159:144]-SRC2[159:144]
DEST[175:160] ← SRC1[175:160]-SRC2[175:160]
DEST[191:176] ← SRC1[191:176]-SRC2[191:176]
DEST[207:192] ← SRC1[207:192]-SRC2[207:192]
DEST[223:208] ← SRC1[223:208]-SRC2[223:208]
DEST[239:224] ← SRC1[239:224]-SRC2[239:224]
DEST[255:240] ← SRC1[255:240]-SRC2[255:240]
```

#### PSUBD (with 64-bit operands)

```

DEST[31:0] ← DEST[31:0] – SRC[31:0];
DEST[63:32] ← DEST[63:32] – SRC[63:32];
```

#### PSUBD (with 128-bit operands)

```

DEST[31:0] ← DEST[31:0] – SRC[31:0];
(* Repeat subtract operation for 2nd and 3rd doubleword *)
DEST[127:96] ← DEST[127:96] – SRC[127:96];
```

#### VPSUBD (VEX.128 encoded version)

```

DEST[31:0] ← SRC1[31:0]-SRC2[31:0]
DEST[63:32] ← SRC1[63:32]-SRC2[63:32]
DEST[95:64] ← SRC1[95:64]-SRC2[95:64]
DEST[127:96] ← SRC1[127:96]-SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

#### VPSUBD (VEX.256 encoded version)

```

DEST[31:0] ← SRC1[31:0]-SRC2[31:0]
DEST[63:32] ← SRC1[63:32]-SRC2[63:32]
DEST[95:64] ← SRC1[95:64]-SRC2[95:64]
DEST[127:96] ← SRC1[127:96]-SRC2[127:96]
DEST[159:128] ← SRC1[159:128]-SRC2[159:128]
DEST[191:160] ← SRC1[191:160]-SRC2[191:160]
DEST[223:192] ← SRC1[223:192]-SRC2[223:192]
DEST[255:224] ← SRC1[255:224]-SRC2[255:224]
```

#### Intel C/C++ Compiler Intrinsic Equivalents

PSUBB:	<code>__m64 _mm_sub_pi8(__m64 m1, __m64 m2)</code>
(V)PSUBB:	<code>__m128i _mm_sub_epi8 ( __m128i a, __m128i b)</code>
VPSUBB:	<code>__m256i _mm256_sub_epi8 ( __m256i a, __m256i b)</code>
PSUBW:	<code>__m64 _mm_sub_pi16(__m64 m1, __m64 m2)</code>
(V)PSUBW:	<code>__m128i _mm_sub_epi16 ( __m128i a, __m128i b)</code>
VPSUBW:	<code>__m256i _mm256_sub_epi16 ( __m256i a, __m256i b)</code>

PSUBD:        $\text{__m64 mm\_sub\_pi32}(\text{__m64 m1}, \text{__m64 m2})$   
(V)PSUBD:      $\text{__m128i mm\_sub\_epi32}(\text{__m128i a}, \text{__m128i b})$   
VPSUBD:        $\text{__m256i mm256\_sub\_epi32}(\text{__m256i a}, \text{__m256i b})$

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD              If VEX.L = 1.

## PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF FB /r <sup>1</sup> PSUBQ mm1, mm2/m64	RM	V/V	SSE2	Subtract quadword integer in <i>mm1</i> from <i>mm2</i> /m64.
66 OF FB /r PSUBQ xmm1, xmm2/m128	RM	V/V	SSE2	Subtract packed quadword integers in <i>xmm1</i> from <i>xmm2</i> /m128.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed quadword integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG FB /r VPSUBQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract packed quadword integers in <i>ymm3/m256</i> from <i>ymm2</i> .

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

### Operation

#### PSUBQ (with 64-Bit operands)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0];$$

**PSUBQ (with 128-Bit operands)**

```
DEST[63:0] ← DEST[63:0] – SRC[63:0];
DEST[127:64] ← DEST[127:64] – SRC[127:64];
```

**VPSUBQ (VEX.128 encoded version)**

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

**VPSUBQ (VEX.256 encoded version)**

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[191:128] ← SRC1[191:128]-SRC2[191:128]
DEST[255:192] ← SRC1[255:192]-SRC2[255:192]
```

**Intel C/C++ Compiler Intrinsic Equivalents**

PSUBQ:	<code>__m64 _mm_sub_si64(__m64 m1, __m64 m2)</code>
(V)PSUBQ:	<code>__m128i _mm_sub_epi64(__m128i m1, __m128i m2)</code>
VPSUBQ:	<code>__m256i _mm256_sub_epi64(__m256i m1, __m256i m2)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF E8 /r <sup>1</sup> PSUBSB <i>mm, mm/m64</i>	RM	V/V	MMX	Subtract signed packed bytes in <i>mm/m64</i> from signed packed bytes in <i>mm</i> and saturate results.
66 OF E8 /r PSUBSB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Subtract packed signed byte integers in <i>xmm2/m128</i> from packed signed byte integers in <i>xmm1</i> and saturate results.
OF E9 /r <sup>1</sup> PSUBSW <i>mm, mm/m64</i>	RM	V/V	MMX	Subtract signed packed words in <i>mm/m64</i> from signed packed words in <i>mm</i> and saturate results.
66 OF E9 /r PSUBSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Subtract packed signed word integers in <i>xmm2/m128</i> from packed signed word integers in <i>xmm1</i> and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Subtract packed signed byte integers in <i>xmm3/m128</i> from packed signed byte integers in <i>xmm2</i> and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Subtract packed signed word integers in <i>xmm3/m128</i> from packed signed word integers in <i>xmm2</i> and saturate results.
VEX.NDS.256.66.0F.WIG E8 /r VPSUBSB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed signed byte integers in <i>ymm3/m256</i> from packed signed byte integers in <i>ymm2</i> and saturate results.
VEX.NDS.256.66.0F.WIG E9 /r VPSUBSW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed signed word integers in <i>ymm3/m256</i> from packed signed word integers in <i>ymm2</i> and saturate results.

### NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE version:** When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

**128-bit Legacy SSE version:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded version:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**VEX.256 encoded version:** The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

## Operation

### PSUBSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC[7:0]);
(* Repeat subtract operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56]);
```

### PSUBSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] – SRC[127:120]);
```

### VPSUBSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

### VPSUBSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
```

### PSUBSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat subtract operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord (DEST[63:48] – SRC[63:48]);
```

### PSUBSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] – SRC[127:112]);
```

### VPSUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

**VPSUBSW (VEX.256 encoded version)**

$\text{DEST}[15:0] \leftarrow \text{SaturateToSignedWord}(\text{SRC1}[15:0] - \text{SRC2}[15:0]);$   
 (\* Repeat subtract operation for 2nd through 15th words \*)  
 $\text{DEST}[255:240] \leftarrow \text{SaturateToSignedWord}(\text{SRC1}[255:240] - \text{SRC2}[255:240]);$

**Intel C/C++ Compiler Intrinsic Equivalents**

PSUBSB:	<code>__m64 _mm_subsb_pi8(__m64 m1, __m64 m2)</code>
(V)PSUBSB:	<code>__m128i _mm_subsb_epi8(__m128i m1, __m128i m2)</code>
VPSUBSB:	<code>__m256i _mm256_subsb_epi8(__m256i m1, __m256i m2)</code>
PSUBSW:	<code>__m64 _mm_subsw_pi16(__m64 m1, __m64 m2)</code>
(V)PSUBSW:	<code>__m128i _mm_subsw_epi16(__m128i m1, __m128i m2)</code>
VPSUBSW:	<code>__m256i _mm256_subsw_epi16(__m256i m1, __m256i m2)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

**PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF D8 / <sup>r</sup> PSUBUSB <i>mm, mm/m64</i>	RM	V/V	MMX	Subtract unsigned packed bytes in <i>mm/m64</i> from unsigned packed bytes in <i>mm</i> and saturate result.
66 OF D8 / <i>r</i> PSUBUSB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Subtract packed unsigned byte integers in <i>xmm2/m128</i> from packed unsigned byte integers in <i>xmm1</i> and saturate result.
OF D9 / <sup>r</sup> PSUBUSW <i>mm, mm/m64</i>	RM	V/V	MMX	Subtract unsigned packed words in <i>mm/m64</i> from unsigned packed words in <i>mm</i> and saturate result.
66 OF D9 / <i>r</i> PSUBUSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Subtract packed unsigned word integers in <i>xmm2/m128</i> from packed unsigned word integers in <i>xmm1</i> and saturate result.
VEX.NDS.128.66.0F.WIG D8 / VPSUBUSB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Subtract packed unsigned byte integers in <i>xmm3/m128</i> from packed unsigned byte integers in <i>xmm2</i> and saturate result.
VEX.NDS.128.66.0F.WIG D9 / VPSUBUSW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Subtract packed unsigned word integers in <i>xmm3/m128</i> from packed unsigned word integers in <i>xmm2</i> and saturate result.
VEX.NDS.256.66.0F.WIG D8 / VPSUBUSB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed unsigned byte integers in <i>ymm3/m256</i> from packed unsigned byte integers in <i>ymm2</i> and saturate result.
VEX.NDS.256.66.0F.WIG D9 / VPSUBUSW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed unsigned word integers in <i>ymm3/m256</i> from packed unsigned word integers in <i>ymm2</i> and saturate result.

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE version:** When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

**128-bit Legacy SSE version:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded version:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**VEX.256 encoded version:** The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

## Operation

### PSUBUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC [7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] – SRC[63:56];
```

### PSUBUSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnSignedByte (DEST[127:120] – SRC[127:120]);
```

### VPSUBUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

### VPSUBUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
```

### PSUBUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] – SRC[63:48] );
```

### PSUBUSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnSignedWord (DEST[127:112] – SRC[127:112]);
```

### VPSUBUSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

**VPSUBUSW (VEX.256 encoded version)**

$\text{DEST}[15:0] \leftarrow \text{SaturateToUnsignedWord}(\text{SRC1}[15:0] - \text{SRC2}[15:0]);$   
 (\* Repeat subtract operation for 2nd through 15th words \*)  
 $\text{DEST}[255:240] \leftarrow \text{SaturateToUnsignedWord}(\text{SRC1}[255:240] - \text{SRC2}[255:240]);$

**Intel C/C++ Compiler Intrinsic Equivalents**

PSUBUSB:  $\text{__m64}_\text{mm\_subs\_pu8}(\text{__m64 m1}, \text{__m64 m2})$   
 (V)PSUBUSB:  $\text{__m128i}_\text{mm\_subs\_epu8}(\text{__m128i m1}, \text{__m128i m2})$   
 VPSUBUSB:  $\text{__m256i}_\text{mm256\_subs\_epu8}(\text{__m256i m1}, \text{__m256i m2})$   
 PSUBUSW:  $\text{__m64}_\text{mm\_subs\_pu16}(\text{__m64 m1}, \text{__m64 m2})$   
 (V)PSUBUSW:  $\text{__m128i}_\text{mm\_subs\_epu16}(\text{__m128i m1}, \text{__m128i m2})$   
 VPSUBUSW:  $\text{__m256i}_\text{mm256\_subs\_epu16}(\text{__m256i m1}, \text{__m256i m2})$

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PTEST- Logical Compare

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 17 /r PTEST <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Set ZF if <i>xmm2/m128 AND xmm1</i> result is all Os. Set CF if <i>xmm2/m128 AND NOT xmm1</i> result is all Os.
VEX.128.66.0F38.WIG 17 /r VPTEST <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.
VEX.256.66.0F38.WIG 17 /r VPTEST <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

### Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

The first source register is specified by the ModR/M *reg* field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

#### (V)PTEST (128-bit version)

```

IF (SRC[127:0] BITWISE AND DEST[127:0] = 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[127:0] BITWISE AND NOT DEST[127:0] = 0)
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

#### VPTEST (VEX.256 encoded version)

```

IF (SRC[255:0] BITWISE AND DEST[255:0] = 0) THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[255:0] BITWISE AND NOT DEST[255:0] = 0) THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

**Intel C/C++ Compiler Intrinsic Equivalent****PTEST**

```
int _mm_testz_si128 (_m128i s1, _m128i s2);
int _mm_testc_si128 (_m128i s1, _m128i s2);
int _mm_testnzc_si128 (_m128i s1, _m128i s2);
```

**VPTEST**

```
int _mm256_testz_si256 (_m256i s1, _m256i s2);
int _mm256_testc_si256 (_m256i s1, _m256i s2);
int _mm256_testnzc_si256 (_m256i s1, _m256i s2);
int _mm_testz_si128 (_m128i s1, _m128i s2);
int _mm_testc_si128 (_m128i s1, _m128i s2);
int _mm_testnzc_si128 (_m128i s1, _m128i s2);
```

**Flags Affected**

The 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 Exceptions Type 4; additionally

#UD                    If VEX.vvvv != 1111B.

**PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 68 / <sup>1</sup> PUNPCKHBW <i>mm, mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order bytes from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 68 / <i>r</i> PUNPCKHBW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 69 / <sup>1</sup> PUNPCKHWD <i>mm, mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order words from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 69 / <i>r</i> PUNPCKHWD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 6A / <sup>1</sup> PUNPCKHDQ <i>mm, mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order doublewords from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 6A / <i>r</i> PUNPCKHDQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 0F 6D / <i>r</i> PUNPCKHQDQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order quadwords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 68/ <i>r</i> VPUNPCKHBW <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 69/ <i>r</i> VPUNPCKHWD <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 6A/ <i>r</i> VPUNPCKHDQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order doublewords from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 6D/ <i>r</i> VPUNPCKHQDQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order quadword from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register.
VEX.NDS.256.66.0F.WIG 68 / <i>r</i> VPUNPCKHBW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order bytes from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 69 / <i>r</i> VPUNPCKHWD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order words from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 6A / <i>r</i> VPUNPCKHDQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order doublewords from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 6D / <i>r</i> VPUNPCKHQDQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order quadword from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.

**NOTES:**

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-16 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

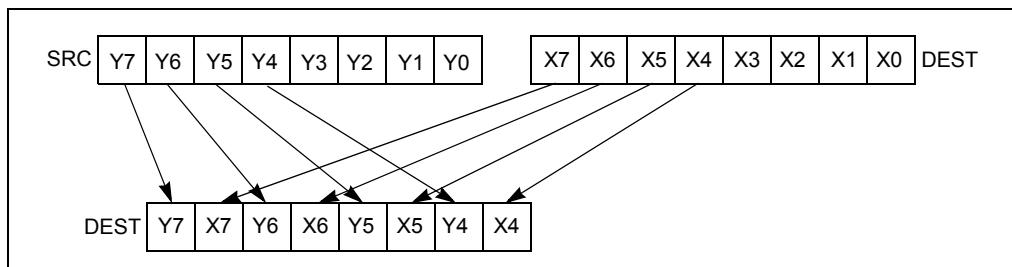


Figure 4-16. PUNPCKHBW Instruction Operation Using 64-bit Operands

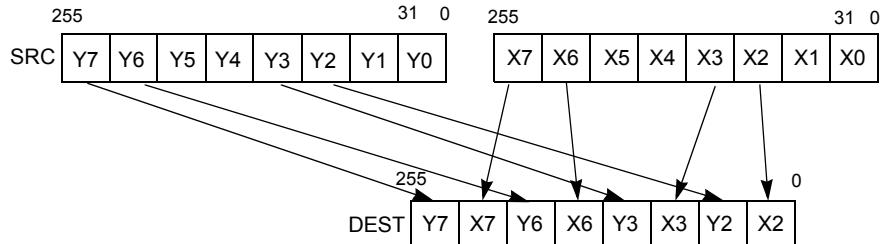


Figure 4-17. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE versions:** The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

**128-bit Legacy SSE versions:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

## Operation

### PUNPCKHBW instruction with 64-bit operands:

```
DEST[7:0] ← DEST[39:32];
DEST[15:8] ← SRC[39:32];
DEST[23:16] ← DEST[47:40];
DEST[31:24] ← SRC[47:40];
DEST[39:32] ← DEST[55:48];
DEST[47:40] ← SRC[55:48];
DEST[55:48] ← DEST[63:56];
DEST[63:56] ← SRC[63:56];
```

### PUNPCKHW instruction with 64-bit operands:

```
DEST[15:0] ← DEST[47:32];
DEST[31:16] ← SRC[47:32];
DEST[47:32] ← DEST[63:48];
DEST[63:48] ← SRC[63:48];
```

### PUNPCKHDQ instruction with 64-bit operands:

```
DEST[31:0] ← DEST[63:32];
DEST[63:32] ← SRC[63:32];
```

### PUNPCKHBW instruction with 128-bit operands:

```
DEST[7:0] ← DEST[71:64];
DEST[15:8] ← SRC[71:64];
DEST[23:16] ← DEST[79:72];
DEST[31:24] ← SRC[79:72];
DEST[39:32] ← DEST[87:80];
DEST[47:40] ← SRC[87:80];
DEST[55:48] ← DEST[95:88];
DEST[63:56] ← SRC[95:88];
DEST[71:64] ← DEST[103:96];
DEST[79:72] ← SRC[103:96];
DEST[87:80] ← DEST[111:104];
DEST[95:88] ← SRC[111:104];
DEST[103:96] ← DEST[119:112];
DEST[111:104] ← SRC[119:112];
DEST[119:112] ← DEST[127:120];
DEST[127:120] ← SRC[127:120];
```

### PUNPCKHWD instruction with 128-bit operands:

```
DEST[15:0] ← DEST[79:64];
DEST[31:16] ← SRC[79:64];
DEST[47:32] ← DEST[95:80];
DEST[63:48] ← SRC[95:80];
DEST[79:64] ← DEST[111:96];
DEST[95:80] ← SRC[111:96];
DEST[111:96] ← DEST[127:112];
```

```
DEST[127:112] ← SRC[127:112];
```

**PUNPCKHDQ instruction with 128-bit operands:**

```
DEST[31:0] ← DEST[95:64];
DEST[63:32] ← SRC[95:64];
DEST[95:64] ← DEST[127:96];
DEST[127:96] ← SRC[127:96];
```

**PUNPCKHQDQ instruction:**

```
DEST[63:0] ← DEST[127:64];
DEST[127:64] ← SRC[127:64];
```

**INTERLEAVE\_HIGH\_BYTES\_256b (SRC1, SRC2)**

```
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]
DEST[135:128] ← SRC1[199:192]
DEST[143:136] ← SRC2[199:192]
DEST[151:144] ← SRC1[207:200]
DEST[159:152] ← SRC2[207:200]
DEST[167:160] ← SRC1[215:208]
DEST[175:168] ← SRC2[215:208]
DEST[183:176] ← SRC1[223:216]
DEST[191:184] ← SRC2[223:216]
DEST[199:192] ← SRC1[231:224]
DEST[207:200] ← SRC2[231:224]
DEST[215:208] ← SRC1[239:232]
DEST[223:216] ← SRC2[239:232]
DEST[231:224] ← SRC1[247:240]
DEST[239:232] ← SRC2[247:240]
DEST[247:240] ← SRC1[255:248]
DEST[255:248] ← SRC2[255:248]
```

**INTERLEAVE\_HIGH\_BYTES (SRC1, SRC2)**

```
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
```

```

DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]

```

**INTERLEAVE\_HIGH\_WORDS\_256b(SRC1, SRC2)**

```

DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]
DEST[143:128] ← SRC1[207:192]
DEST[159:144] ← SRC2[207:192]
DEST[175:160] ← SRC1[223:208]
DEST[191:176] ← SRC2[223:208]
DEST[207:192] ← SRC1[239:224]
DEST[223:208] ← SRC2[239:224]
DEST[239:224] ← SRC1[255:240]
DEST[255:240] ← SRC2[255:240]

```

**INTERLEAVE\_HIGH\_WORDS (SRC1, SRC2)**

```

DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]

```

**INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1, SRC2)**

```

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]

```

**INTERLEAVE\_HIGH\_DWORDS(SRC1, SRC2)**

```

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]

```

**INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1, SRC2)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[127:64]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC2}[127:64]$   
 $\text{DEST}[191:128] \leftarrow \text{SRC1}[255:192]$   
 $\text{DEST}[255:192] \leftarrow \text{SRC2}[255:192]$

**INTERLEAVE\_HIGH\_QWORDS(SRC1, SRC2)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[127:64]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC2}[127:64]$

**PUNPCKHBW (128-bit Legacy SSE Version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_HIGH\_BYTES}(\text{DEST}, \text{SRC})$   
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

**VPUNPCKHBW (VEX.128 encoded version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_HIGH\_BYTES}(\text{SRC1}, \text{SRC2})$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VPUNPCKHBW (VEX.256 encoded version)**

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE\_HIGH\_BYTES\_256b}(\text{SRC1}, \text{SRC2})$

**PUNPCKHWD (128-bit Legacy SSE Version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_HIGH\_WORDS}(\text{DEST}, \text{SRC})$   
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

**VPUNPCKHWD (VEX.128 encoded version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_HIGH\_WORDS}(\text{SRC1}, \text{SRC2})$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VPUNPCKHWD (VEX.256 encoded version)**

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE\_HIGH\_WORDS\_256b}(\text{SRC1}, \text{SRC2})$

**PUNPCKHDQ (128-bit Legacy SSE Version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_HIGH\_DWORDS}(\text{DEST}, \text{SRC})$   
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

**VPUNPCKHDQ (VEX.128 encoded version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_HIGH\_DWORDS}(\text{SRC1}, \text{SRC2})$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VPUNPCKHDQ (VEX.256 encoded version)**

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE\_HIGH\_DWORDS\_256b}(\text{SRC1}, \text{SRC2})$

**PUNPCKHQDQ (128-bit Legacy SSE Version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_HIGH\_QWORDS}(\text{DEST}, \text{SRC})$   
 $\text{DEST}[255:127] \text{ (Unmodified)}$

**VPUNPCKHQDQ (VEX.128 encoded version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_HIGH\_QWORDS}(\text{SRC1}, \text{SRC2})$   
 $\text{DEST}[255:127] \leftarrow 0$

**VPUNPCKHQDQ (VEX.256 encoded version)**

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE\_HIGH\_QWORDS\_256}(\text{SRC1}, \text{SRC2})$

## Intel C/C++ Compiler Intrinsic Equivalents

PUNPCKHBW:	<code>_m64_mm_unpackhi_pi8(_m64 m1, _m64 m2)</code>
(V)PUNPCKHBW:	<code>_m128i_mm_unpackhi_epi8(_m128i m1, _m128i m2)</code>
VPUNPCKHBW:	<code>_m256i_mm256_unpackhi_epi8(_m256i m1, _m256i m2)</code>
PUNPCKHWD:	<code>_m64_mm_unpackhi_pi16(_m64 m1, _m64 m2)</code>
(V)PUNPCKHWD:	<code>_m128i_mm_unpackhi_epi16(_m128i m1, _m128i m2)</code>
VPUNPCKHWD:	<code>_m256i_mm256_unpackhi_epi16(_m256i m1, _m256i m2)</code>
PUNPCKHDQ:	<code>_m64_mm_unpackhi_pi32(_m64 m1, _m64 m2)</code>
(V)PUNPCKHDQ:	<code>_m128i_mm_unpackhi_epi32(_m128i m1, _m128i m2)</code>
VPUNPCKHDQ:	<code>_m256i_mm256_unpackhi_epi32(_m256i m1, _m256i m2)</code>
(V)PUNPCKHQDQ:	<code>_m128i_mm_unpackhi_epi64 (_m128i a, _m128i b)</code>
VPUNPCKHQDQ:	<code>_m256i_mm256_unpackhi_epi64 (_m256i a, _m256i b)</code>

## Flags Affected

None.

## Numeric Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

**PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 bit Mode Support</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
OF 60 /r <sup>1</sup> PUNPCKLBW <i>mm, mm/m32</i>	RM	V/V	MMX	Interleave low-order bytes from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 60 /r PUNPCKLBW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
OF 61 /r <sup>1</sup> PUNPCKLWD <i>mm, mm/m32</i>	RM	V/V	MMX	Interleave low-order words from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 61 /r PUNPCKLWD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
OF 62 /r <sup>1</sup> PUNPCKLDQ <i>mm, mm/m32</i>	RM	V/V	MMX	Interleave low-order doublewords from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 62 /r PUNPCKLDQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 OF 6C /r PUNPCKLQDQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order quadword from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> register.
VEX.NDS.128.66.0F.WIG 60/r VPUNPCKLBW <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 61/r VPUNPCKLWD <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 62/r VPUNPCKLDQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order doublewords from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 6C/r VPUNPCKLQDQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order quadword from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register.
VEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order bytes from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order words from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 62 /r VPUNPCKLDQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order doublewords from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 6C /r VPUNPCKLQDQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order quadword from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.

**NOTES:**

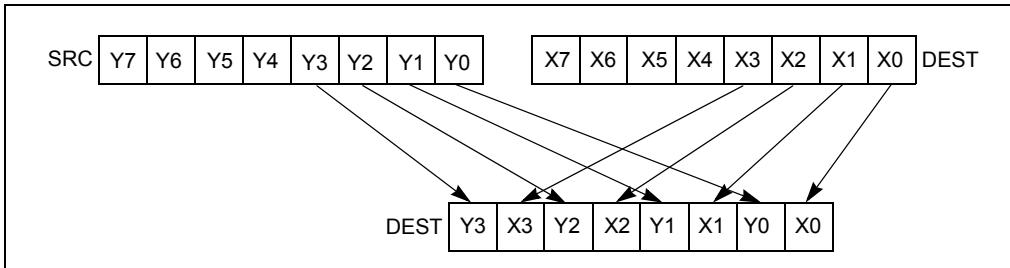
1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

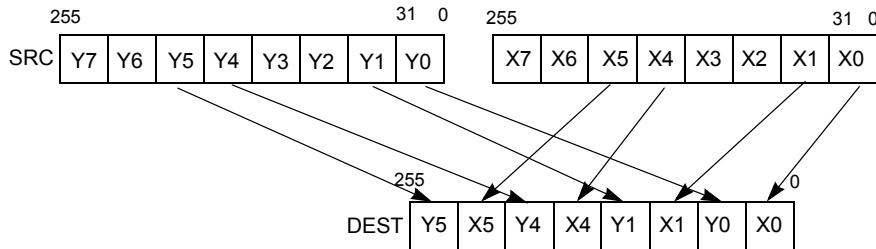
Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-18 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.



**Figure 4-18. PUNPCKLBW Instruction Operation Using 64-bit Operands**



**Figure 4-19. 256-bit VPUNPCKLDQ Instruction Operation**

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE versions:** The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

**128-bit Legacy SSE versions:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded versions:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

## Operation

### PUNPCKLBW instruction with 64-bit operands:

```
DEST[63:56] ← SRC[31:24];
DEST[55:48] ← DEST[31:24];
DEST[47:40] ← SRC[23:16];
DEST[39:32] ← DEST[23:16];
DEST[31:24] ← SRC[15:8];
DEST[23:16] ← DEST[15:8];
DEST[15:8] ← SRC[7:0];
DEST[7:0] ← DEST[7:0];
```

### PUNPCKLWD instruction with 64-bit operands:

```
DEST[63:48] ← SRC[31:16];
DEST[47:32] ← DEST[31:16];
DEST[31:16] ← SRC[15:0];
DEST[15:0] ← DEST[15:0];
```

### PUNPCKLDQ instruction with 64-bit operands:

```
DEST[63:32] ← SRC[31:0];
DEST[31:0] ← DEST[31:0];
```

### PUNPCKLBW instruction with 128-bit operands:

```
DEST[7:0] ← DEST[7:0];
DEST[15:8] ← SRC[7:0];
DEST[23:16] ← DEST[15:8];
DEST[31:24] ← SRC[15:8];
DEST[39:32] ← DEST[23:16];
DEST[47:40] ← SRC[23:16];
DEST[55:48] ← DEST[31:24];
DEST[63:56] ← SRC[31:24];
DEST[71:64] ← DEST[39:32];
DEST[79:72] ← SRC[39:32];
DEST[87:80] ← DEST[47:40];
DEST[95:88] ← SRC[47:40];
DEST[103:96] ← DEST[55:48];
DEST[111:104] ← SRC[55:48];
DEST[119:112] ← DEST[63:56];
DEST[127:120] ← SRC[63:56];
```

### PUNPCKLWD instruction with 128-bit operands:

```
DEST[15:0] ← DEST[15:0];
DEST[31:16] ← SRC[15:0];
DEST[47:32] ← DEST[31:16];
DEST[63:48] ← SRC[31:16];
DEST[79:64] ← DEST[47:32];
DEST[95:80] ← SRC[47:32];
DEST[111:96] ← DEST[63:48];
DEST[127:112] ← SRC[63:48];
```

**PUNPCKLDQ instruction with 128-bit operands:**

```
DEST[31:0] ← DEST[31:0];
DEST[63:32] ← SRC[31:0];
DEST[95:64] ← DEST[63:32];
DEST[127:96] ← SRC[63:32];
```

**PUNPCKLQDQ**

```
DEST[63:0] ← DEST[63:0];
DEST[127:64] ← SRC[63:0];
```

**INTERLEAVE\_BYTES\_256b (SRC1, SRC2)**

```
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC1[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]
DEST[135:128] ← SRC1[135:128]
DEST[143:136] ← SRC2[135:128]
DEST[151:144] ← SRC1[143:136]
DEST[159:152] ← SRC2[143:136]
DEST[167:160] ← SRC1[151:144]
DEST[175:168] ← SRC2[151:144]
DEST[183:176] ← SRC1[159:152]
DEST[191:184] ← SRC2[159:152]
DEST[199:192] ← SRC1[167:160]
DEST[207:200] ← SRC2[167:160]
DEST[215:208] ← SRC1[175:168]
DEST[223:216] ← SRC2[175:168]
DEST[231:224] ← SRC1[183:176]
DEST[239:232] ← SRC2[183:176]
DEST[247:240] ← SRC1[191:184]
DEST[255:248] ← SRC2[191:184]
```

**INTERLEAVE\_BYTES (SRC1, SRC2)**

```
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC2[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
```

## INSTRUCTION SET REFERENCE, N-Z

```
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]
```

### **INTERLEAVE\_WORDS\_256b(SRC1, SRC2)**

```
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]
DEST[143:128] ← SRC1[143:128]
DEST[159:144] ← SRC2[143:128]
DEST[175:160] ← SRC1[159:144]
DEST[191:176] ← SRC2[159:144]
DEST[207:192] ← SRC1[175:160]
DEST[223:208] ← SRC2[175:160]
DEST[239:224] ← SRC1[191:176]
DEST[255:240] ← SRC2[191:176]
```

### **INTERLEAVE\_WORDS (SRC1, SRC2)**

```
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]
```

### **INTERLEAVE\_DWORDS\_256b(SRC1, SRC2)**

```
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]
DEST[191:160] ← SRC2[159:128]
DEST[223:192] ← SRC1[191:160]
DEST[255:224] ← SRC2[191:160]
```

### **INTERLEAVE\_DWORDS(SRC1, SRC2)**

```
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
```

### **INTERLEAVE\_QWORDS\_256b(SRC1, SRC2)**

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
```

$\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128]$   
 $\text{DEST}[255:192] \leftarrow \text{SRC2}[191:128]$

**PUNPCKLBW (128-bit Legacy SSE Version)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC2}[63:0]$

**PUNPCKLBW (128-bit Legacy SSE Version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_BYTES}(\text{DEST}, \text{SRC})$   
 $\text{DEST}[255:127] \text{ (Unmodified)}$

**VPUNPCKLBW (VEX.128 encoded instruction)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_BYTES}(\text{SRC1}, \text{SRC2})$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VPUNPCKLBW (VEX.256 encoded instruction)**

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE\_BYTES\_128b}(\text{SRC1}, \text{SRC2})$

**PUNPCKLWD (128-bit Legacy SSE Version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_WORDS}(\text{DEST}, \text{SRC})$   
 $\text{DEST}[255:127] \text{ (Unmodified)}$

**VPUNPCKLWD (VEX.128 encoded instruction)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_WORDS}(\text{SRC1}, \text{SRC2})$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VPUNPCKLWD (VEX.256 encoded instruction)**

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE\_WORDS}(\text{SRC1}, \text{SRC2})$

**PUNPCKLDQ (128-bit Legacy SSE Version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_DWORDS}(\text{DEST}, \text{SRC})$   
 $\text{DEST}[255:127] \text{ (Unmodified)}$

**VPUNPCKLDQ (VEX.128 encoded instruction)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_DWORDS}(\text{SRC1}, \text{SRC2})$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VPUNPCKLDQ (VEX.256 encoded instruction)**

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE\_DWORDS}(\text{SRC1}, \text{SRC2})$

**PUNPCKLQDQ (128-bit Legacy SSE Version)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_QWORDS}(\text{DEST}, \text{SRC})$   
 $\text{DEST}[255:127] \text{ (Unmodified)}$

**VPUNPCKLQDQ (VEX.128 encoded instruction)**

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE\_QWORDS}(\text{SRC1}, \text{SRC2})$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VPUNPCKLQDQ (VEX.256 encoded instruction)**

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE\_QWORDS}(\text{SRC1}, \text{SRC2})$

**Intel C/C++ Compiler Intrinsic Equivalents**

PUNPCKLBW:       $\_m64 \_mm\_unpacklo\_pi8 (\_m64 m1, \_m64 m2)$

(V)PUNPCKLBW:  $\text{__m128i\_mm\_unpacklo\_epi8}(\text{__m128i m1}, \text{__m128i m2})$   
 VPUNPCKLBW:  $\text{__m256i\_mm256\_unpacklo\_epi8}(\text{__m256i m1}, \text{__m256i m2})$   
 PUNPCKLWD:  $\text{__m64\_mm\_unpacklo\_pi16}(\text{__m64 m1}, \text{__m64 m2})$   
 (V)PUNPCKLWD:  $\text{__m128i\_mm\_unpacklo\_epi16}(\text{__m128i m1}, \text{__m128i m2})$   
 VPUNPCKLWD:  $\text{__m256i\_mm256\_unpacklo\_epi16}(\text{__m256i m1}, \text{__m256i m2})$   
 PUNPCKLDQ:  $\text{__m64\_mm\_unpacklo\_pi32}(\text{__m64 m1}, \text{__m64 m2})$   
 (V)PUNPCKLDQ:  $\text{__m128i\_mm\_unpacklo\_epi32}(\text{__m128i m1}, \text{__m128i m2})$   
 VPUNPCKLDQ:  $\text{__m256i\_mm256\_unpacklo\_epi32}(\text{__m256i m1}, \text{__m256i m2})$   
 (V)PUNPCKLQDQ:  $\text{__m128i\_mm\_unpacklo\_epi64}(\text{__m128i m1}, \text{__m128i m2})$   
 VPUNPCKLQDQ:  $\text{__m256i\_mm256\_unpacklo\_epi64}(\text{__m256i m1}, \text{__m256i m2})$

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 1.

## PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH <i>r/m16</i>	M	Valid	Valid	Push <i>r/m16</i> .
FF /6	PUSH <i>r/m32</i>	M	N.E.	Valid	Push <i>r/m32</i> .
FF /6	PUSH <i>r/m64</i>	M	Valid	N.E.	Push <i>r/m64</i> .
50+rw	PUSH <i>r16</i>	O	Valid	Valid	Push <i>r16</i> .
50+rd	PUSH <i>r32</i>	O	N.E.	Valid	Push <i>r32</i> .
50+rd	PUSH <i>r64</i>	O	Valid	N.E.	Push <i>r64</i> .
6A ib	PUSH <i>imm8</i>	I	Valid	Valid	Push <i>imm8</i> .
68 iw	PUSH <i>imm16</i>	I	Valid	Valid	Push <i>imm16</i> .
68 id	PUSH <i>imm32</i>	I	Valid	Valid	Push <i>imm32</i> .
0E	PUSH CS	NP	Invalid	Valid	Push CS.
16	PUSH SS	NP	Invalid	Valid	Push SS.
1E	PUSH DS	NP	Invalid	Valid	Push DS.
06	PUSH ES	NP	Invalid	Valid	Push ES.
OF A0	PUSH FS	NP	Valid	Valid	Push FS.
OF A8	PUSH GS	NP	Valid	Valid	Push GS.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA
O	opcode + rd (r)	NA	NA	NA
I	imm8/16/32	NA	NA	NA
NP	NA	NA	NA	NA

### Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

The address size is used only when referencing a source operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

## Operation

(\* See Description section for possible sign-extension or zero-extension of source operand and for \*)

(\* a case in which the size of the memory store may be smaller than the instruction's operand size \*)

IF StackAddrSize = 64

```

THEN
IF OperandSize = 64
THEN
    RSP ← RSP - 8;
    Memory[SS:RSP] ← SRC; (* push quadword *)
ELSE IF OperandSize = 32
THEN
    RSP ← RSP - 4;
    Memory[SS:RSP] ← SRC; (* push dword *)
ELSE (* OperandSize = 16 *)
    RSP ← RSP - 2;
    Memory[SS:RSP] ← SRC; (* push word *)
FI;

SE IF StackAddrSize = 32
THEN
IF OperandSize = 64
THEN
    ESP ← ESP - 8;
    Memory[SS:ESP] ← SRC; (* push quadword *)
ELSE IF OperandSize = 32
THEN
    ESP ← ESP - 4;
    Memory[SS:ESP] ← SRC; (* push dword *)
ELSE (* OperandSize = 16 *)
    ESP ← ESP - 2;
    Memory[SS:ESP] ← SRC; (* push word *)
FI;
ELSE (* StackAddrSize = 16 *)

```

```

IF OperandSize = 32
THEN
    SP ← SP - 4;
    Memory[SS:SP] ← SRC;          (* push dword *)
ELSE (* OperandSize = 16 *)
    SP ← SP - 2;
    Memory[SS:SP] ← SRC;          (* push word *)
FI;
FI;

```

## Flags Affected

None.

## Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
	If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used. If the PUSH is of CS, SS, DS, or ES.

## PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
60	PUSHA	NP	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	NP	Invalid	Valid	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the “Operation” section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

IF 64-bit Mode

    THEN #UD

FI;

IF OperandSize = 32 (\* PUSHAD instruction \*)

    THEN

```
        Temp ← (ESP);
        Push(EAX);
        Push(ECX);
        Push(EDX);
        Push(EBX);
        Push(Temp);
        Push(EBP);
        Push(ESI);
        Push(EDI);
```

ELSE (\* OperandSize = 16, PUSHA instruction \*)

```
        Temp ← (SP);
        Push(AX);
        Push(CX);
        Push(DX);
```

```

Push(BX);
Push(Temp);
Push(BP);
Push(SI);
Push(DI);
Fl;

```

### Flags Affected

None.

### Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#UD	If in 64-bit mode.
-----	--------------------

**PUSHF/PUSHFD—Push EFLAGS Register onto the Stack**

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9C	PUSHF	NP	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	NP	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	NP	Valid	N.E.	Push RFLAGS.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

**Description**

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFD instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction’s default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When in virtual-8086 mode and the I/O privilege level (IOPL) is less than 3, the PUSHF/PUSHFD instruction causes a general protection exception (#GP).

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Operation**

IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))

(\* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 \*)

THEN

  IF OperandSize = 32

    THEN

      push (EFLAGS AND 00FCFFFFH);

      (\* VM and RF EFLAG bits are cleared in image stored on the stack \*)

    ELSE

      push (EFLAGS); (\* Lower 16 bits only \*)

  FI;

ELSE IF 64-bit MODE (\* In 64-bit Mode \*)

  IF OperandSize = 64

```

THEN
    push (RFLAGS AND 00000000_00FCFFFFH);
    (* VM and RF RFLAG bits are cleared in image stored on the stack; *)
ELSE
    push (EFLAGS); (* Lower 16 bits only *)
FI;

ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
    #GP(0); (* Trap to virtual-8086 monitor *)
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

### Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

## PXOR—Logical Exclusive OR

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF EF /r <sup>1</sup> PXOR mm, mm/m64	RM	V/V	MMX	Bitwise XOR of mm/m64 and mm.
66 OF EF /r PXOR xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise XOR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.
VEX.NDS.256.66.0F.WIG EF /r VPXOR ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise XOR of ymm3/m256 and ymm2.

### NOTES:

- See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

### Operation

#### PXOR (128-bit Legacy SSE version)

DEST  $\leftarrow$  DEST XOR SRC  
DEST[VLMAX-1:128] (Unmodified)

#### VPXOR (VEX.128 encoded version)

DEST  $\leftarrow$  SRC1 XOR SRC2  
DEST[VLMAX-1:128]  $\leftarrow$  0

**VPXOR (VEX.256 encoded version)**DEST  $\leftarrow$  SRC1 XOR SRC2**Intel C/C++ Compiler Intrinsic Equivalent**

PXOR:       `__m64 _mm_xor_si64 (__m64 m1, __m64 m2)`  
(V)PXOR:     `__m128i _mm_xor_si128 ( __m128i a, __m128i b)`  
VPXOR:       `__m256i _mm256_xor_si256 ( __m256i a, __m256i b)`

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD              If VEX.L = 1.

**RCL/RCR/ROL/ROR—Rotate**

<b>Opcode**</b>	<b>Instruction</b>	<b>Op/ En</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
D0 /2	RCL r/m8, 1	M1	Valid	Valid	Rotate 9 bits (CF, r/m8) left once.
REX + D0 /2	RCL r/m8*, 1	M1	Valid	N.E.	Rotate 9 bits (CF, r/m8) left once.
D2 /2	RCL r/m8, CL	MC	Valid	Valid	Rotate 9 bits (CF, r/m8) left CL times.
REX + D2 /2	RCL r/m8*, CL	MC	Valid	N.E.	Rotate 9 bits (CF, r/m8) left CL times.
C0 /2 ib	RCL r/m8, imm8	MI	Valid	Valid	Rotate 9 bits (CF, r/m8) left <i>imm8</i> times.
REX + C0 /2 ib	RCL r/m8*, imm8	MI	Valid	N.E.	Rotate 9 bits (CF, r/m8) left <i>imm8</i> 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 <i>imm8</i> 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 <i>imm8</i> times.
REX.W + C1 /2 ib	RCL r/m64, imm8	MI	Valid	N.E.	Rotate 65 bits (CF, r/m64) left <i>imm8</i> times. Uses a 6 bit count.
D0 /3	RCR r/m8, 1	M1	Valid	Valid	Rotate 9 bits (CF, r/m8) right once.
REX + D0 /3	RCR r/m8*, 1	M1	Valid	N.E.	Rotate 9 bits (CF, r/m8) right once.
D2 /3	RCR r/m8, CL	MC	Valid	Valid	Rotate 9 bits (CF, r/m8) right CL times.
REX + D2 /3	RCR r/m8*, CL	MC	Valid	N.E.	Rotate 9 bits (CF, r/m8) right CL times.
C0 /3 ib	RCR r/m8, imm8	MI	Valid	Valid	Rotate 9 bits (CF, r/m8) right <i>imm8</i> times.
REX + C0 /3 ib	RCR r/m8*, imm8	MI	Valid	N.E.	Rotate 9 bits (CF, r/m8) right <i>imm8</i> 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 <i>imm8</i> 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 <i>imm8</i> times.
REX.W + C1 /3 ib	RCR r/m64, imm8	MI	Valid	N.E.	Rotate 65 bits (CF, r/m64) right <i>imm8</i> times. Uses a 6 bit count.
D0 /0	ROL r/m8, 1	M1	Valid	Valid	Rotate 8 bits r/m8 left once.
REX + D0 /0	ROL r/m8*, 1	M1	Valid	N.E.	Rotate 8 bits r/m8 left once
D2 /0	ROL r/m8, CL	MC	Valid	Valid	Rotate 8 bits r/m8 left CL times.
REX + D2 /0	ROL r/m8*, CL	MC	Valid	N.E.	Rotate 8 bits r/m8 left CL times.
C0 /0 ib	ROL r/m8, imm8	MI	Valid	Valid	Rotate 8 bits r/m8 left <i>imm8</i> times.

Opcode**	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX + CO /0 ib	ROL r/m8*, imm8	MI	Valid	N.E.	Rotate 8 bits r/m8 left imm8 times.
D1 /0	ROL r/m16, 1	M1	Valid	Valid	Rotate 16 bits r/m16 left once.
D3 /0	ROL r/m16, CL	MC	Valid	Valid	Rotate 16 bits r/m16 left CL times.
C1 /0 ib	ROL r/m16, imm8	MI	Valid	Valid	Rotate 16 bits r/m16 left imm8 times.
D1 /0	ROL r/m32, 1	M1	Valid	Valid	Rotate 32 bits r/m32 left once.
REX.W + D1 /0	ROL r/m64, 1	M1	Valid	N.E.	Rotate 64 bits r/m64 left once. Uses a 6 bit count.
D3 /0	ROL r/m32, CL	MC	Valid	Valid	Rotate 32 bits r/m32 left CL times.
REX.W + D3 /0	ROL r/m64, CL	MC	Valid	N.E.	Rotate 64 bits r/m64 left CL times. Uses a 6 bit count.
C1 /0 ib	ROL r/m32, imm8	MI	Valid	Valid	Rotate 32 bits r/m32 left imm8 times.
REX.W + C1 /0 ib	ROL r/m64, imm8	MI	Valid	N.E.	Rotate 64 bits r/m64 left imm8 times. Uses a 6 bit count.
D0 /1	ROR r/m8, 1	M1	Valid	Valid	Rotate 8 bits r/m8 right once.
REX + D0 /1	ROR r/m8*, 1	M1	Valid	N.E.	Rotate 8 bits r/m8 right once.
D2 /1	ROR r/m8, CL	MC	Valid	Valid	Rotate 8 bits r/m8 right CL times.
REX + D2 /1	ROR r/m8*, CL	MC	Valid	N.E.	Rotate 8 bits r/m8 right CL times.
CO /1 ib	ROR r/m8, imm8	MI	Valid	Valid	Rotate 8 bits r/m16 right imm8 times.
REX + CO /1 ib	ROR r/m8*, imm8	MI	Valid	N.E.	Rotate 8 bits r/m16 right imm8 times.
D1 /1	ROR r/m16, 1	M1	Valid	Valid	Rotate 16 bits r/m16 right once.
D3 /1	ROR r/m16, CL	MC	Valid	Valid	Rotate 16 bits r/m16 right CL times.
C1 /1 ib	ROR r/m16, imm8	MI	Valid	Valid	Rotate 16 bits r/m16 right imm8 times.
D1 /1	ROR r/m32, 1	M1	Valid	Valid	Rotate 32 bits r/m32 right once.
REX.W + D1 /1	ROR r/m64, 1	M1	Valid	N.E.	Rotate 64 bits r/m64 right once. Uses a 6 bit count.
D3 /1	ROR r/m32, CL	MC	Valid	Valid	Rotate 32 bits r/m32 right CL times.
REX.W + D3 /1	ROR r/m64, CL	MC	Valid	N.E.	Rotate 64 bits r/m64 right CL times. Uses a 6 bit count.
C1 /1 ib	ROR r/m32, imm8	MI	Valid	Valid	Rotate 32 bits r/m32 right imm8 times.
REX.W + C1 /1 ib	ROR r/m64, imm8	MI	Valid	N.E.	Rotate 64 bits r/m64 right imm8 times. Uses a 6 bit count.

**NOTES:**

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

\*\* See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (w)	1	NA	NA
MC	ModRM:r/m (w)	CL	NA	NA
MI	ModRM:r/m (w)	imm8	NA	NA

## Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. In legacy and compatibility mode, the processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

## IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

## Operation

(\* RCL and RCR instructions \*)

```
SIZE ← OperandSize;
CASE (determine count) OF
  SIZE ← 8:  tempCOUNT ← (COUNT AND 1FH) MOD 9;
  SIZE ← 16: tempCOUNT ← (COUNT AND 1FH) MOD 17;
  SIZE ← 32: tempCOUNT ← COUNT AND 1FH;
  SIZE ← 64: tempCOUNT ← COUNT AND 3FH;
ESAC;
```

(\* RCL instruction operation \*)

```
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← MSB(DEST);
    DEST ← (DEST * 2) + CF;
    CF ← tempCF;
    tempCOUNT ← tempCOUNT - 1;
  OD;
  ELIHW;
  IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
  FI;
```

(\* RCR instruction operation \*)

IF COUNT = 1

```

THEN OF ← MSB(DEST) XOR CF;
ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
DO
    tempCF ← LSB(SRC);
    DEST ← (DEST / 2) + (CF * 2SIZE);
    CF ← tempCF;
    tempCOUNT ← tempCOUNT - 1;
OD;

(* ROL and ROR instructions *)
IF OperandSize = 64
    THEN COUNTMASK = 3FH;
    ELSE COUNTMASK = 1FH;
FI;

(* ROL instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE

WHILE (tempCOUNT ≠ 0)
DO
    tempCF ← MSB(DEST);
    DEST ← (DEST * 2) + tempCF;
    tempCOUNT ← tempCOUNT - 1;
OD;
ELIHw;
CF ← LSB(DEST);
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;

(* ROR instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT ≠ 0)
DO
    tempCF ← LSB(SRC);
    DEST ← (DEST / 2) + (tempCF * 2SIZE);
    tempCOUNT ← tempCOUNT - 1;
OD;
ELIHw;
CF ← MSB(DEST);
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR MSB - 1(DEST);
    ELSE OF is undefined;
FI;
```

### Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see “Description” above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

### Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the source operand is located in a nonwritable segment.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 53 /r RCPPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Computes the approximate reciprocals of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.0F.WIG 53 /r VRCPPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in <i>xmm2/mem</i> and stores the results in <i>xmm1</i> .
VEX.256.0F.WIG 53 /r VRCPPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in <i>ymm2/mem</i> and stores the results in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to  $|1.1111111110100000000000B * 2^{125}|$  are guaranteed to not produce tiny results; input values less than or equal to  $|1.00000000000110000000001B * 2^{126}|$  are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

## Operation

### RCPPS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[VLMAX-1:128] (Unmodified)
```

### VRCPPS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[VLMAX-1:128] ← 0
```

### VRCPPS (VEX.256 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[159:128] ← APPROXIMATE(1/SRC[159:128])
DEST[191:160] ← APPROXIMATE(1/SRC[191:160])
DEST[223:192] ← APPROXIMATE(1/SRC[223:192])
DEST[255:224] ← APPROXIMATE(1/SRC[255:224])
```

## Intel C/C++ Compiler Intrinsic Equivalent

RCCPS:	<code>__m128 _mm_rcp_ps(__m128 a)</code>
RCPPS:	<code>__m256 _mm256_rcp_ps (__m256 a);</code>

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.
-----	-----------------------

## RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 53 /r RCPSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Computes the approximate reciprocal of the scalar single-precision floating-point value in <i>xmm2/m32</i> and stores the result in <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.WIG 53 /r VRCPSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Computes the approximate reciprocal of the scalar single-precision floating-point value in <i>xmm3/m32</i> and stores the result in <i>xmm1</i> . Also, upper single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes of an approximate reciprocal of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to  $|1.1111111101000000000B * 2^{125}|$  are guaranteed to not produce tiny results; input values less than or equal to  $|1.00000000001100000001B * 2^{126}|$  are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

#### RCPSS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])  
DEST[VLMAX-1:32] (Unmodified)

#### VRCPSS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SRC2[31:0])  
DEST[127:32] ← SRC1[127:32]  
DEST[VLMAX-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

RCPSS:       `__m128 _mm_rcp_ss(__m128 a)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5.

## RDFSBASE/RDGSBASE—Read FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Fea- ture Flag	Description
F3 OF AE /0 RDFSBASE r32	M	V/I	FSGSBASE	Load the 32-bit destination register with the FS base address.
REX.W + F3 OF AE /0 RDFSBASE r64	M	V/I	FSGSBASE	Load the 64-bit destination register with the FS base address.
F3 OF AE /1 RDGSBASE r32	M	V/I	FSGSBASE	Load the 32-bit destination register with the GS base address.
REX.W + F3 OF AE /1 RDGSBASE r64	M	V/I	FSGSBASE	Load the 64-bit destination register with the GS base address.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Loads the general-purpose register indicated by the modR/M:r/m field with the FS or GS segment base address.

The destination operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source base address (for FS or GS) are ignored and upper 32 bits of the destination register are cleared.

This instruction is supported only in 64-bit mode.

### Operation

DEST ← FS/GS segment base address;

### Flags Affected

None

### C/C++ Compiler Intrinsic Equivalent

RDFSBASE:	unsigned int _readfsbase_u32(void);
RDFSBASE:	unsigned __int64 _readfsbase_u64(void);
RDGSBASE:	unsigned int _readgsbase_u32(void);
RDGSBASE:	unsigned __int64 _readgsbase_u64(void);

### Protected Mode Exceptions

#UD               The RDFSBASE and RDGSBASE instructions are not recognized in protected mode.

### Real-Address Mode Exceptions

#UD               The RDFSBASE and RDGSBASE instructions are not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD               The RDFSBASE and RDGSBASE instructions are not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD                The RDGSBASE and RDGSBASE instructions are not recognized in compatibility mode.

### 64-Bit Mode Exceptions

#UD                If the LOCK prefix is used.

If CR4.FSGSBASE[bit 16] = 0.

If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0.

## RDMSR—Read from Model Specific Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 32	RDMSR	NP	Valid	Valid	Read MSR specified by ECX into EDX:EAX.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Chapter 35, “Model-Specific Registers (MSRs),” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

EDX:EAX ← MSR[ECX];

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- If the value in ECX specifies a reserved or unimplemented MSR address.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP If the value in ECX specifies a reserved or unimplemented MSR address.
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- #GP(0) The RDMSR instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## RDPMC—Read Performance-Monitoring Counters

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 33	RDPMC	NP	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

The EAX register is loaded with the low-order 32 bits. The EDX register is loaded with the supported high-order bits of the counter. The number of high-order bits loaded into EDX is implementation specific on processors that do not support architectural performance monitoring. The width of fixed-function and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID OAH leaf. See below for the treatment of the EDX register for “fast” reads.

The ECX register selects one of two type of performance counters, specifies the index relative to the base of each counter type, and selects “fast” read mode if supported. The two counter types are :

- General-purpose or special-purpose performance counters: The number of general-purpose counters is model specific if the processor does not support architectural performance monitoring, see Chapter 30 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. Special-purpose counters are available only in selected processor members, see Section 30.13, 30.14 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. This counter type is selected if ECX[30] is clear.
- Fixed-function performance counter. The number fixed-function performance counters is enumerated by CPUID OAH leaf. See Chapter 30 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. This counter type is selected if ECX[30] is set.

ECX[29:0] specifies the index. The width of general-purpose performance counters are 40-bits for processors that do not support architectural performance monitoring counters. The width of special-purpose performance counters are implementation specific. The width of fixed-function performance counters and general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID OAH leaf.

Table 4-12 lists valid indices of the general-purpose and special-purpose performance counters according to the derived DisplayFamily\_DisplayModel values of CPUID encoding for each processor family (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-M” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*).

**Table 4-12. Valid General and Special Purpose Performance Counter Index Range for RDPMC**

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
P6	06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_OAH, 06H_0BH	0, 1	0, 1
Pentium® 4, Intel® Xeon processors	0FH_00H, 0FH_01H, 0FH_02H	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium 4, Intel Xeon processors	(0FH_03H, 0FH_04H, 0FH_06H) and (L3 is absent)	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium M processors	06H_09H, 06H_0DH	0, 1	0, 1
64-bit Intel Xeon processors with L3	0FH_03H, 0FH_04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™ Solo and Intel® Core™ Duo processors, Dual-core Intel® Xeon® processor LV	06H_0EH	0, 1	0, 1

**Table 4-12. Valid General and Special Purpose Performance Counter Index Range for RDPMC (Contd.)**

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300, 7300 Series - general-purpose PMC	06H_0FH	0, 1	0, 1
Intel Xeon processors 7100 series with L3	(0FH_06H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™2 Duo processor family, Intel Xeon processor family - general-purpose PMC	06H_17H	0, 1	0, 1
Intel Xeon processors 7400 series	(06H_1DH)	≥ 0 and ≤ 9	0, 1
Intel® Atom™ processor family	06H_1CH	0, 1	0, 1
Intel® Core™i7 processor, Intel Xeon processors 5500 series	06H_1AH, 06H_1EH, 06H_1FH, 06H_2EH	0-3	0, 1, 2, 3

The Pentium 4 and Intel Xeon processors also support “fast” (32-bit) and “slow” (40-bit) reads on the first 18 performance counters. Selected this option using ECX[31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on Pentium 4 processors and Intel Xeon processors than a full 40-bit read.

On 64-bit Intel Xeon processors with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters. On Intel Xeon processor 7100 series with L3, performance counters with indices 18-25 are also 32-bit counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, 5300 and 7400 series, the fixed-function performance counters are 40-bits wide; they can be accessed by RDPMC with ECX between from 4000\_0000H and 4000\_0002H.

On Intel Xeon processor 7400 series, there are eight 32-bit special-purpose counters addressable with indices 2-9, ECX[30]=0.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Chapter 19, “Performance Monitoring Events,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

In the Pentium 4 and Intel Xeon processors, performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

## Operation

(\* Intel Core i7 processor family and Intel Xeon processor 3400, 5500 series\*)

Most significant counter bit (MSCB) = 47

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CRO.PE = 0))
  THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
    EAX ← IA32_FIXED_CTR(ECX)[30:0];
    EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0];
    EDX ← PMC(ECX[30:0])[MSCB:32];
  ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CRO.PE is 1 *)
    #GP(0);
FI;
```

(\* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300, 7400 series\*)

Most significant counter bit (MSCB) = 39

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CRO.PE = 0))
  THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
    EAX ← IA32_FIXED_CTR(ECX)[30:0];
    EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0];
    EDX ← PMC(ECX[30:0])[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid special-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
  ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CRO.PE is 1 *)
    #GP(0);
FI;
```

(\* P6 family processors and Pentium processor with MMX technology \*)

```
IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CRO.PE = 0))
  THEN
    EAX ← PMC(ECX)[31:0];
    EDX ← PMC(ECX)[39:32];
  ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CRO.PE is 1 *)
    #GP(0);
FI;
```

(\* Processors with CPUID family 15 \*)

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CRO.PE = 0))
  THEN IF (ECX[30:0] = 0:17)
    THEN IF ECX[31] = 0
      THEN
        EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
        EDX ← PMC(ECX[30:0])[39:32];
      ELSE (* ECX[31] = 1*)
        THEN
          EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
          EDX ← 0;
        FI;
      ELSE IF (*64-bit Intel Xeon processor with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
```

```

EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
EDX ← 0;
Fl;
ELSE IF (*Intel Xeon processor 7100 series with L3 *)
    THEN IF (ECX[30:0] = 18:25 )
        EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
        EDX ← 0;
    Fl;
ELSE (* Invalid PMC index in ECX[30:0], see Table 4-15. *)
    GP(0);
Fl;
ELSE (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CRO.PE = 1 *)
    #GP(0);
Fl;

```

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified (see Table 4-12).  
(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.
- #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

- #GP If an invalid performance counter index is specified (see Table 4-12).  
(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.
- #UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

- #GP(0) If the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified (see Table 4-12).  
(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.
- #UD If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified in ECX[30:0] (see Table 4-12).
- #UD If the LOCK prefix is used.

## RDRAND—Read Random Number

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C7 /6 RDRAND r16	M	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
OF C7 /6 RDRAND r32	M	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
REX.W + OF C7 /6 RDRAND r64	M	V/I	RDRAND	Read a 64-bit random number and store in the destination register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND (see *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, Section 7.3.17, “Random Number Generator Instruction”).

This instruction is available at all privilege levels.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```

IF HW_RND_GEN.ready = 1
  THEN
    CASE of
      osize is 64: DEST[63:0] ← HW_RND_GEN.data;
      osize is 32: DEST[31:0] ← HW_RND_GEN.data;
      osize is 16: DEST[15:0] ← HW_RND_GEN.data;
    ESAC
    CF ← 1;
  ELSE
    CASE of
      osize is 64: DEST[63:0] ← 0;
      osize is 32: DEST[31:0] ← 0;
      osize is 16: DEST[15:0] ← 0;
    ESAC
    CF ← 0;
  FI
OF, SF, ZF, AF, PF ← 0;
```

### Flags Affected

The CF flag is set according to the result (see the “Operation” section above). The OF, SF, ZF, AF, and PF flags are set to 0.

### Intel C/C++ Compiler Intrinsic Equivalent

```
RDRAND:     int _rdrand16_step( unsigned short * );
RDRAND:     int _rdrand32_step( unsigned int * );
RDRAND:     int _rdrand64_step( unsigned __int64 * );
```

### Protected Mode Exceptions

#UD                  If the LOCK prefix is used.  
                      If the F2H or F3H prefix is used.  
                      If CPUID.01H:ECX.RDRAND[bit 30] = 0.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## RDTSC—Read Time-Stamp Counter

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 31	RDTSC	NP	Valid	Valid	Read time-stamp counter into EDX:EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Loads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

When in protected or virtual 8086 mode, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the TSD 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. (When in real-address mode, the RDTSC instruction is always enabled.)

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. If software requires RDTSC to be executed only after all previous instructions have completed locally, it can either use RDTSCP (if the processor supports that instruction) or execute the sequence LFENCE;RDTSC.

This instruction was introduced by the Pentium processor.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CRO.PE = 0)
    THEN EDX:EAX ← TimeStampCounter;
ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CRO.PE = 1 *)
    #GP(0);
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

**Virtual-8086 Mode Exceptions**

- #GP(0) If the TSD flag in register CR4 is set.
- #UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## RDTSCP—Read Time-Stamp Counter and Processor ID

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 F9	RDTSCP	NP	Valid	Valid	Read 64-bit time-stamp counter and 32-bit IA32_TSC_AUX value into EDX:EAX and ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Loads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also loads the IA32\_TSC\_AUX MSR (address C000\_0103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32\_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32\_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32\_TSC\_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

When in protected or virtual 8086 mode, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the TSD 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. (When in real-address mode, the RDTSCP instruction is always enabled.)

The RDTSCP instruction waits until all previous instructions have been executed before reading the counter. However, subsequent instructions may begin execution before the read operation is performed.

The presence of the RDTSCP instruction is indicated by CPUID leaf 80000001H, EDX bit 27. If the bit is set to 1 then RDTSCP is present on the processor.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

```

IF (CR4.TSD = 0) or (CPL = 0) or (CRO.PE = 0)
  THEN
    EDX:EAX ← TimeStampCounter;
    ECX ← IA32_TSC_AUX[31:0];
  ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CRO.PE = 1 *)
    #GP(0);
  Fl;

```

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.
- #UD If the LOCK prefix is used.
- If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

### Real-Address Mode Exceptions

- #UD                  If the LOCK prefix is used.  
                        If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

### Virtual-8086 Mode Exceptions

- #GP(0)              If the TSD flag in register CR4 is set.
- #UD                  If the LOCK prefix is used.  
                        If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 6C	REP INS <i>m8</i> , DX	NP	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS <i>m8</i> , DX	NP	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS <i>m16</i> , DX	NP	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI].
F3 6D	REP INS <i>m32</i> , DX	NP	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS <i>r/m32</i> , DX	NP	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS <i>m8, m8</i>	NP	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A4	REP MOVS <i>m8, m8</i>	NP	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS <i>m16, m16</i>	NP	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS <i>m32, m32</i>	NP	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A5	REP MOVS <i>m64, m64</i>	NP	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS DX, <i>r/m8</i>	NP	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 REX.W 6E	REP OUTS DX, <i>r/m8*</i>	NP	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS DX, <i>r/m16</i>	NP	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, <i>r/m32</i>	NP	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 REX.W 6F	REP OUTS DX, <i>r/m32</i>	NP	Valid	N.E.	Output RCX default size from [RSI] to port DX.
F3 AC	REP LODS AL	NP	Valid	Valid	Load (E)CX bytes from DS:[(E)SI] to AL.
F3 REX.W AC	REP LODS AL	NP	Valid	N.E.	Load RCX bytes from [RSI] to AL.
F3 AD	REP LODS AX	NP	Valid	Valid	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS EAX	NP	Valid	Valid	Load (E)CX doublewords from DS:[(E)SI] to EAX.
F3 REX.W AD	REP LODS RAX	NP	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.
F3 AA	REP STOS <i>m8</i>	NP	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 REX.W AA	REP STOS <i>m8</i>	NP	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS <i>m16</i>	NP	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS <i>m32</i>	NP	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 REX.W AB	REP STOS <i>m64</i>	NP	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS <i>m8, m8</i>	NP	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A6	REPE CMPS <i>m8, m8</i>	NP	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS <i>m16, m16</i>	NP	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS <i>m32, m32</i>	NP	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A7	REPE CMPS <i>m64, m64</i>	NP	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS <i>m8</i>	NP	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
F3 REX.W AE	REPE SCAS <i>m8</i>	NP	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS <i>m16</i>	NP	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS <i>m32</i>	NP	Valid	Valid	Find non-EAX doubleword starting at ES:[(E)DI].

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 REX.W AF	REPE SCAS <i>m64</i>	NP	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS <i>m8, m8</i>	NP	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A6	REPNE CMPS <i>m8, m8</i>	NP	Valid	N.E.	Find matching bytes in [RDI] and [RSI].
F2 A7	REPNE CMPS <i>m16, m16</i>	NP	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS <i>m32, m32</i>	NP	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A7	REPNE CMPS <i>m64, m64</i>	NP	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].
F2 AE	REPNE SCAS <i>m8</i>	NP	Valid	Valid	Find AL, starting at ES:[(E)DI].
F2 REX.W AE	REPNE SCAS <i>m8</i>	NP	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS <i>m16</i>	NP	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS <i>m32</i>	NP	Valid	Valid	Find EAX, starting at ES:[(E)DI].
F2 REX.W AF	REPNE SCAS <i>m64</i>	NP	Valid	N.E.	Find RAX, starting at [RDI].

**NOTES:**

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

**Description**

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string 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. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-13.

**Table 4-13. Repeat Prefixes**

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

**NOTES:**

\* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.

The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, the operand size of the count register is associated with the address size attribute. Thus the default count register is RCX; REX.W has no effect on the address size and the count register. In 64-bit mode, if 67H is used to override address size attribute, the count register is ECX and any implicit source/destination operand will use the corresponding 32-bit index register. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```

IF AddressSize = 16
  THEN
    Use CX for CountReg;
    Implicit Source/Dest operand for memory use of SI/DI;
ELSE IF AddressSize = 64
  THEN Use RCX for CountReg;
  Implicit Source/Dest operand for memory use of RSI/RDI;
ELSE
  Use ECX for CountReg;
  Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
  DO
    Service pending interrupts (if any);
    Execute associated string instruction;
    CountReg ← (CountReg - 1);
    IF CountReg = 0
      THEN exit WHILE loop; FI;
    IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
    or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
      THEN exit WHILE loop; FI;
OD;
```

## Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

## Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
--------	---

## RET—Return from Procedure

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	NP	Valid	Valid	Near return to calling procedure.
CB	RET	NP	Valid	Valid	Far return to calling procedure.
C2 <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Near return to calling procedure and pop <i>imm16</i> bytes from stack.
CA <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Far return to calling procedure and pop <i>imm16</i> bytes from stack.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
I	imm16	NA	NA	NA

### Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure’s stack and the calling procedure’s stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits.

## 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); Fl;
        EIP ← Pop();
      ELSE
        IF OperandSize = 64
          THEN
            IF top 8 bytes of stack not within stack limits
              THEN #SS(0); Fl;
              RIP ← Pop();
            ELSE (* OperandSize = 16 *)
              IF top 2 bytes of stack not within stack limits
                THEN #SS(0); Fl;
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits
                  THEN #GP(0); Fl;
                EIP ← tempEIP;
              Fl;
            Fl;
          Fl;

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;
          Fl;
        Fl;
      Fl;
    Fl;

(* 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); Fl;
          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); Fl;

```

## INSTRUCTION SET REFERENCE, N-Z

```
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;
        FI;
        IF return code segment selector is NULL
            THEN #GP(0); FI;
        IF return code segment selector addresses descriptor beyond descriptor table limit
            THEN #GP(selector); FI;
        Obtain descriptor to which return code segment selector points from descriptor table;
        IF return code segment descriptor is not a code segment
            THEN #GP(selector); FI;
        IF return code segment selector RPL < CPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is conforming
            and return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is non-conforming and return code
            segment DPL ≠ return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is not present
            THEN #NP(selector); FI;
        IF return code segment selector RPL > CPL
            THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
            ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
        FI;
    FI;

RETURN-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
```

```

ELSE (* OperandSize = 16 *)
    EIP ← Pop();
    EIP ← EIP AND 0000FFFFH;
    CS ← Pop(); (* 16-bit pop *)
FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
        FI;
    FI;

RETURN-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
Read return segment selector;
IF stack segment selector is NULL
    THEN #GP(0); FI;
IF return stack segment selector index is not within its descriptor table limits
    THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                    ELSE (* StackAddressSize = 16 *)
                        SP ← SP + SRC;
                FI;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
        FI;
    FI;

```

## INSTRUCTION SET REFERENCE, N-Z

```
EIP ← EIP AND 0000FFFFH;
CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
CS(RPL) ← CPL;
IF instruction has immediate operand
    THEN (* Release parameters from called procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
            FI;
        FI;
        tempESP ← Pop();
        tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
        ESP ← tempESP;
        SS ← tempSS;
    FI;

FOR each of segment register (ES, FS, GS, and DS)
DO
    IF segment register points to data or non-conforming code segment
    and CPL > segment descriptor DPL (* DPL in hidden part of segment register *)
        THEN SegmentSelector ← 0; (* Segment selector invalid *)
    FI;
OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
            FI;
        FI;
(* IA-32e Mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
THEN
    IF OperandSize = 32
        THEN
            IF second doubleword on stack is not within stack limits
                THEN #SS(0); FI;
            IF first or second doubleword on stack is not in canonical space
                THEN #SS(0); FI;
        ELSE
            IF OperandSize = 16
                THEN
                    IF second word on stack is not within stack limits
                        THEN #SS(0); FI;
                    IF first or second word on stack is not in canonical space
                        THEN #SS(0); FI;
                ELSE (* OperandSize = 64 *)
                    IF first or second quadword on stack is not in canonical space
                        THEN #SS(0); FI;
                FI;
            FI;
        FI;
    FI;
```

```

        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;
    IF return code segment selector addresses descriptor in non-canonical space
        THEN GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment descriptor has L-bit = 1 and D-bit = 1
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming
    and return code segment DPL ≠ return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
        THEN GOTO IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

```

IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL:

```

IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        FI;
    FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN

```

## INSTRUCTION SET REFERENCE, N-Z

```
    ESP ← ESP + SRC;
ELSE
    IF StackAddressSize = 16
        THEN
            SP ← SP + SRC;
        ELSE (* StackAddressSize = 64 *)
            RSP ← RSP + SRC;
    FI;
FI;
IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
    FI;
IF return stack segment descriptor is not within descriptor table limits
    THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
    THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                ELSE
```

```

IF StackAddressSize = 16
    THEN
        SP ← SP + SRC;
    ELSE (* StackAddressSize = 64 *)
        RSP ← RSP + SRC;
    FI;
FI;
tempESP ← Pop();
tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
ESP ← tempESP;
SS ← tempSS;
ELSE
    IF OperandSize = 16
        THEN
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP + SRC;
                        ELSE
                            IF StackAddressSize = 16
                                THEN
                                    SP ← SP + SRC;
                                ELSE (* StackAddressSize = 64 *)
                                    RSP ← RSP + SRC;
                                FI;
                            FI;
                            tempESP ← Pop();
                            tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                            ESP ← tempESP;
                            SS ← tempSS;
                        ELSE (* OperandSize = 64 *)
                            RIP ← Pop();
                            CS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
                            CS(RPL) ← CPL;
                            IF instruction has immediate operand
                                THEN (* Release parameters from called procedure's stack *)
                                    RSP ← RSP + SRC;
                                FI;
                            tempESP ← Pop();
                            tempSS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
                            ESP ← tempESP;
                            SS ← tempSS;
                        FI;
                    FI;
    FOR each of segment register (ES, FS, GS, and DS)
        DO

```

```

IF segment register points to data or non-conforming code segment
and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
    THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
FI;
OD;

IF instruction has immediate operand
THEN (* Release parameters from calling procedure's stack *)
    IF StackAddressSize = 32
        THEN
            ESP ← ESP + SRC;
        ELSE
            IF StackAddressSize = 16
                THEN
                    SP ← SP + SRC;
                ELSE (* StackAddressSize = 64 *)
                    RSP ← RSP + SRC;
            FI;
    FI;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector NULL. If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less than the CPL. If the return code or stack segment selector index is not within its descriptor table limits. If the return code segment descriptor does not indicate a code segment. If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector If the stack segment is not a writable data segment. If the stack segment selector RPL is not equal to the RPL of the return code segment selector. If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits. If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

### Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

### Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
--------	---

#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

### Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

### 64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical. If the return instruction pointer is not within the return code segment limit. If the stack segment selector is NULL going back to compatibility mode. If the stack segment selector is NULL going back to CPL3 64-bit mode. If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode. If the return code segment selector is NULL.
#GP(selector)	If the proposed segment descriptor for a code segment does not indicate it is a code segment. If the proposed new code segment descriptor has both the D-bit and L-bit set. If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector. If CPL is greater than the RPL of the code segment selector. If the DPL of a conforming-code segment is greater than the return code segment selector RPL. If a segment selector index is outside its descriptor table limits. If a segment descriptor memory address is non-canonical. If the stack segment is not a writable data segment. If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector. If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit. If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## RORX – Rotate Right Logical Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F3A.W0 F0 /r ib RORX r32, r/m32, imm8	RMI	V/V	BMI2	Rotate 32-bit <i>r/m32</i> right <i>imm8</i> times without affecting arithmetic flags.
VEX.LZ.F2.0F3A.W1 F0 /r ib RORX r64, r/m64, imm8	RMI	V/N.E.	BMI2	Rotate 64-bit <i>r/m64</i> right <i>imm8</i> times without affecting arithmetic flags.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Rotates the bits of second operand right by the count value specified in imm8 without affecting arithmetic flags. The RORX instruction does not read or write the arithmetic flags.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```

IF (OperandSize = 32)
    y ← imm8 AND 1FH;
    DEST ← (SRC >> y) | (SRC << (32-y));
ELSEIF (OperandSize = 64 )
    y ← imm8 AND 3FH;
    DEST ← (SRC >> y) | (SRC << (64-y));
ENDIF

```

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD                    If VEX.W = 1.

## ROUNDPD – Round Packed Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 09 /r ib ROUNDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Round packed double precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.128.66.0F3A.WIG 09 /r ib VROUNDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Round packed double-precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.256.66.0F3A.WIG 09 /r ib VROUNDPD <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX	Round packed double-precision floating-point values in <i>ymm2/m256</i> and place the result in <i>ymm1</i> . The rounding mode is determined by <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

Round the 2 double-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-14 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

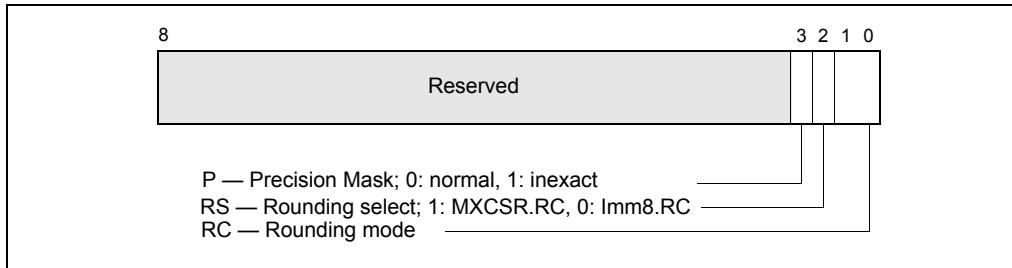


Figure 4-20. Bit Control Fields of Immediate Byte for ROUNDxx Instruction

Table 4-14. 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 $-\infty$ )	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward $+\infty$ )	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

## Operation

```

IF (imm[2] = '1')
    THEN // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_Imm(SRC[127:64]);
FI

```

### ROUNDPD (128-bit Legacy SSE version)

```

DEST[63:0] ← RoundToInteger(SRC[63:0]), ROUND_CONTROL
DEST[127:64] ← RoundToInteger(SRC[127:64]), ROUND_CONTROL
DEST[VLMAX-1:128] (Unmodified)

```

### VROUNDPD (VEX.128 encoded version)

```

DEST[63:0] ← RoundToInteger(SRC[63:0]), ROUND_CONTROL
DEST[127:64] ← RoundToInteger(SRC[127:64]), ROUND_CONTROL
DEST[VLMAX-1:128] ← 0

```

### VROUNDPD (VEX.256 encoded version)

```

DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]), ROUND_CONTROL
DEST[191:128] ← RoundToInteger(SRC[191:128]), ROUND_CONTROL
DEST[255:192] ← RoundToInteger(SRC[255:192]), ROUND_CONTROL

```

## Intel C/C++ Compiler Intrinsic Equivalent

```
__m128 _mm_round_pd(__m128d s1, int iRoundMode);
```

```
__m128 _mm_floor_pd(__m128d s1);
__m128 _mm_ceil_pd(__m128d s1)
__m256 _mm256_round_pd(__m256d s1, int iRoundMode);
__m256 _mm256_floor_pd(__m256d s1);
__m256 _mm256_ceil_pd(__m256d s1)
```

### SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUND PD.

### Other Exceptions

See Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

**ROUNDPS – Round Packed Single Precision Floating-Point Values**

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 08 /r ib ROUNDPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Round packed single precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.128.66.0F3A.WIG 08 /r ib VROUNDPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Round packed single-precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.256.66.0F3A.WIG 08 /r ib VROUNDPS <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX	Round packed single-precision floating-point values in <i>ymm2/m256</i> and place the result in <i>ymm1</i> . The rounding mode is determined by <i>imm8</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

**Description**

Round the 4 single-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-14 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

## Operation

```

IF (imm[2] = '1)
    THEN // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_M(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_M(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_Imm(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_Imm(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;

```

### ROUNDPS(128-bit Legacy SSE version)

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]), ROUND_CONTROL
DEST[127:96] ← RoundToInteger(SRC[127:96]), ROUND_CONTROL
DEST[VLMAX-1:128] (Unmodified)

```

### VROUNDPS (VEX.128 encoded version)

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]), ROUND_CONTROL
DEST[127:96] ← RoundToInteger(SRC[127:96]), ROUND_CONTROL
DEST[VLMAX-1:128] ← 0

```

### VROUNDPS (VEX.256 encoded version)

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]), ROUND_CONTROL
DEST[127:96] ← RoundToInteger(SRC[127:96]), ROUND_CONTROL
DEST[159:128] ← RoundToInteger(SRC[159:128]), ROUND_CONTROL
DEST[191:160] ← RoundToInteger(SRC[191:160]), ROUND_CONTROL
DEST[223:192] ← RoundToInteger(SRC[223:192]), ROUND_CONTROL
DEST[255:224] ← RoundToInteger(SRC[255:224]), ROUND_CONTROL

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

__m128 _mm_round_ps(__m128 s1, int iRoundMode);
__m128 _mm_floor_ps(__m128 s1);
__m128 _mm_ceil_ps(__m128 s1)
__m256 _mm256_round_ps(__m256 s1, int iRoundMode);
__m256 _mm256_floor_ps(__m256 s1);
__m256 _mm256_ceil_ps(__m256 s1)

```

### SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

### Other Exceptions

See Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

## ROUNDSD – Round Scalar Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 3A 0B /r ib ROUNDSD <i>xmm1, xmm2/m64, imm8</i>	RMI	V/V	SSE4_1	Round the low packed double precision floating-point value in <i>xmm2/m64</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.NDS.LIG.66.0F3A.WIG 0B /r ib VROUNDSD <i>xmm1, xmm2, xmm3/m64, imm8</i>	RVMI	V/V	AVX	Round the low packed double precision floating-point value in <i>xmm3/m64</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> . Upper packed double precision floating-point value (bits[127:64]) from <i>xmm2</i> is copied to <i>xmm1</i> [127:64].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Round the DP FP value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double-precision floating-point input to an integer value and returns the integer result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-14 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

```
IF (imm[2] = '1')
  THEN // rounding mode is determined by MXCSR.RC
    DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
  ELSE // rounding mode is determined by IMM8.RC
    DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;
```

### ROUNDSD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[VLMAX-1:64] (Unmodified)
```

**VROUNDSD (VEX.128 encoded version)**

$\text{DEST}[63:0] \leftarrow \text{RoundToInteger}(\text{SRC2}[63:0], \text{ROUND\_CONTROL})$   
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

```
ROUNDSD:   __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
           __m128d mm_floor_sd(__m128d dst, __m128d s1);
           __m128d mm_ceil_sd(__m128d dst, __m128d s1);
```

**SIMD Floating-Point Exceptions**

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSD.

**Other Exceptions**

See Exceptions Type 3.

## ROUNDSS – Round Scalar Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 3A 0A /r ib ROUNDSS <i>xmm1, xmm2/m32, imm8</i>	RMI	V/V	SSE4_1	Round the low packed single precision floating-point value in <i>xmm2/m32</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.NDS.LIG.66.0F3A.WIG 0A /r ib VROUNDSS <i>xmm1, xmm2, xmm3/m32, imm8</i>	RVMI	V/V	AVX	Round the low packed single precision floating-point value in <i>xmm3/m32</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> . Also, upper packed single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1[127:32]</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Round the single-precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single-precision floating-point input to an integer value and returns the result as a single-precision floating-point value in the lowest position. The upper three single-precision floating-point values in the destination are retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-14 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

```

IF (imm[2] = '1)
    THEN // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPTointeger_M(SRC[31:0]);
    ELSE // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPTointeger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged ;

```

### ROUNDSS (128-bit Legacy SSE version)

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[VLMAX-1:32] (Unmodified)

```

**VROUNDSS (VEX.128 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{RoundToInteger}(\text{SRC2}[31:0], \text{ROUND\_CONTROL})$   
 $\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

```
ROUNDSS:   __m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);
            __m128 mm_floor_ss(__m128 dst, __m128 s1);
            __m128 mm_ceil_ss(__m128 dst, __m128 s1);
```

**SIMD Floating-Point Exceptions**

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSS.

**Other Exceptions**

See Exceptions Type 3.

## RSM—Resume from System Management Mode

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AA	RSM	NP	Invalid	Valid	Resume operation of interrupted program.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CRO, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode.

See Chapter 34, "System Management Mode," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about SMM and the behavior of the RSM instruction.

### Operation

```

ReturnFromSMM;
IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_0CH )
    THEN
        ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));
    Else
        ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
FI

```

### Flags Affected

All.

### Protected Mode Exceptions

#UD                  If an attempt is made to execute this instruction when the processor is not in SMM.  
                         If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 52 /r RSQRTPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.0F.WIG 52 /r VRSQRTPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in <i>xmm2/mem</i> and stores the results in <i>xmm1</i> .
VEX.256.0F.WIG 52 /r VRSQRTPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in <i>ymm2/mem</i> and stores the results in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than -0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

## Operation

### RSQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[VLMAX-1:128] (Unmodified)
```

### VRSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[VLMAX-1:128] ← 0
```

### VRSQRTPS (VEX.256 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[159:128] ← APPROXIMATE(1/SQRT(SRC2[159:128]))
DEST[191:160] ← APPROXIMATE(1/SQRT(SRC2[191:160]))
DEST[223:192] ← APPROXIMATE(1/SQRT(SRC2[223:192]))
DEST[255:224] ← APPROXIMATE(1/SQRT(SRC2[255:224]))
```

## Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS:   `__m128 _mm_rsqrt_ps(__m128 a)`  
 RSQRTPS:   `__m256 _mm256_rsqrt_ps (__m256 a);`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD                         If VEX.vvvv != 1111B.

## RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 52 /r RSQRTSS <i>xmm1, xmm2/m32</i>	RM	V/V	SSE	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in <i>xmm2/m32</i> and stores the results in <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.WIG 52 /r VRSQRTSS <i>xmm1, xmm2, xmm3/m32</i>	RVM	V/V	AVX	Computes the approximate reciprocal of the square root of the low single precision floating-point value in <i>xmm3/m32</i> and stores the results in <i>xmm1</i> . Also, upper single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than -0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

#### RSQRTSS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))  
DEST[VLMAX-1:32] (Unmodified)

#### VRSQRTSS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))  
DEST[127:32] ← SRC1[31:0]  
DEST[VLMAX-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

RSQRTSS: `__m128 _mm_rsqrt_ss(__m128 a)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5.

## SAHF—Store AH into Flags

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9E	SAHF	NP	Invalid*	Valid	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register.

### NOTES:

\* Valid in specific steppings. See Description section.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the “Operation” section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

### Operation

```

IF IA-64 Mode
  THEN
    IF CPUID.80000001H(ECX[0]) = 1;
      THEN
        RFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
      ELSE
        #UD;
      FI
    ELSE
      EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
    FI;
  
```

### Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

### Protected Mode Exceptions

None.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

None.

### Compatibility Mode Exceptions

None.

**64-Bit Mode Exceptions**

#UD                    If CPUID.80000001H.ECX[0] = 0.  
                      If the LOCK prefix is used.

**SAL/SAR/SHL/SHR—Shift**

<b>Opcode***</b>	<b>Instruction</b>	<b>Op/ En</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
D0 /4	SAL r/m8, 1	M1	Valid	Valid	Multiply r/m8 by 2, once.
REX + D0 /4	SAL r/m8**, 1	M1	Valid	N.E.	Multiply r/m8 by 2, once.
D2 /4	SAL r/m8, CL	MC	Valid	Valid	Multiply r/m8 by 2, CL times.
REX + D2 /4	SAL r/m8**, CL	MC	Valid	N.E.	Multiply r/m8 by 2, CL times.
C0 /4 ib	SAL r/m8, imm8	MI	Valid	Valid	Multiply r/m8 by 2, imm8 times.
REX + C0 /4 ib	SAL r/m8**, imm8	MI	Valid	N.E.	Multiply r/m8 by 2, imm8 times.
D1 /4	SAL r/m16, 1	M1	Valid	Valid	Multiply r/m16 by 2, once.
D3 /4	SAL r/m16, CL	MC	Valid	Valid	Multiply r/m16 by 2, CL times.
C1 /4 ib	SAL r/m16, imm8	MI	Valid	Valid	Multiply r/m16 by 2, imm8 times.
D1 /4	SAL r/m32, 1	M1	Valid	Valid	Multiply r/m32 by 2, once.
REX.W + D1 /4	SAL r/m64, 1	M1	Valid	N.E.	Multiply r/m64 by 2, once.
D3 /4	SAL r/m32, CL	MC	Valid	Valid	Multiply r/m32 by 2, CL times.
REX.W + D3 /4	SAL r/m64, CL	MC	Valid	N.E.	Multiply r/m64 by 2, CL times.
C1 /4 ib	SAL r/m32, imm8	MI	Valid	Valid	Multiply r/m32 by 2, imm8 times.
REX.W + C1 /4 ib	SAL r/m64, imm8	MI	Valid	N.E.	Multiply r/m64 by 2, imm8 times.
D0 /7	SAR r/m8, 1	M1	Valid	Valid	Signed divide* r/m8 by 2, once.
REX + D0 /7	SAR r/m8**, 1	M1	Valid	N.E.	Signed divide* r/m8 by 2, once.
D2 /7	SAR r/m8, CL	MC	Valid	Valid	Signed divide* r/m8 by 2, CL times.
REX + D2 /7	SAR r/m8**, CL	MC	Valid	N.E.	Signed divide* r/m8 by 2, CL times.
C0 /7 ib	SAR r/m8, imm8	MI	Valid	Valid	Signed divide* r/m8 by 2, imm8 time.
REX + C0 /7 ib	SAR r/m8**, imm8	MI	Valid	N.E.	Signed divide* r/m8 by 2, imm8 times.
D1 /7	SAR r/m16, 1	M1	Valid	Valid	Signed divide* r/m16 by 2, once.
D3 /7	SAR r/m16, CL	MC	Valid	Valid	Signed divide* r/m16 by 2, CL times.
C1 /7 ib	SAR r/m16, imm8	MI	Valid	Valid	Signed divide* r/m16 by 2, imm8 times.
D1 /7	SAR r/m32, 1	M1	Valid	Valid	Signed divide* r/m32 by 2, once.
REX.W + D1 /7	SAR r/m64, 1	M1	Valid	N.E.	Signed divide* r/m64 by 2, once.
D3 /7	SAR r/m32, CL	MC	Valid	Valid	Signed divide* r/m32 by 2, CL times.
REX.W + D3 /7	SAR r/m64, CL	MC	Valid	N.E.	Signed divide* r/m64 by 2, CL times.
C1 /7 ib	SAR r/m32, imm8	MI	Valid	Valid	Signed divide* r/m32 by 2, imm8 times.
REX.W + C1 /7 ib	SAR r/m64, imm8	MI	Valid	N.E.	Signed divide* r/m64 by 2, imm8 times
D0 /4	SHL r/m8, 1	M1	Valid	Valid	Multiply r/m8 by 2, once.
REX + D0 /4	SHL r/m8**, 1	M1	Valid	N.E.	Multiply r/m8 by 2, once.
D2 /4	SHL r/m8, CL	MC	Valid	Valid	Multiply r/m8 by 2, CL times.
REX + D2 /4	SHL r/m8**, CL	MC	Valid	N.E.	Multiply r/m8 by 2, CL times.
C0 /4 ib	SHL r/m8, imm8	MI	Valid	Valid	Multiply r/m8 by 2, imm8 times.
REX + C0 /4 ib	SHL r/m8**, imm8	MI	Valid	N.E.	Multiply r/m8 by 2, imm8 times.
D1 /4	SHL r/m16, 1	M1	Valid	Valid	Multiply r/m16 by 2, once.
D3 /4	SHL r/m16, CL	MC	Valid	Valid	Multiply r/m16 by 2, CL times.
C1 /4 ib	SHL r/m16, imm8	MI	Valid	Valid	Multiply r/m16 by 2, imm8 times.
D1 /4	SHL r/m32, 1	M1	Valid	Valid	Multiply r/m32 by 2, once.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + D1 /4	SHL r/m64,1	M1	Valid	N.E.	Multiply r/m64 by 2, once.
D3 /4	SHL r/m32, CL	MC	Valid	Valid	Multiply r/m32 by 2, CL times.
REX.W + D3 /4	SHL r/m64, CL	MC	Valid	N.E.	Multiply r/m64 by 2, CL times.
C1 /4 ib	SHL r/m32, imm8	MI	Valid	Valid	Multiply r/m32 by 2, imm8 times.
REX.W + C1 /4 ib	SHL r/m64, imm8	MI	Valid	N.E.	Multiply r/m64 by 2, imm8 times.
D0 /5	SHR r/m8,1	M1	Valid	Valid	Unsigned divide r/m8 by 2, once.
REX + D0 /5	SHR r/m8**, 1	M1	Valid	N.E.	Unsigned divide r/m8 by 2, once.
D2 /5	SHR r/m8, CL	MC	Valid	Valid	Unsigned divide r/m8 by 2, CL times.
REX + D2 /5	SHR r/m8**, CL	MC	Valid	N.E.	Unsigned divide r/m8 by 2, CL times.
C0 /5 ib	SHR r/m8, imm8	MI	Valid	Valid	Unsigned divide r/m8 by 2, imm8 times.
REX + C0 /5 ib	SHR r/m8**, imm8	MI	Valid	N.E.	Unsigned divide r/m8 by 2, imm8 times.
D1 /5	SHR r/m16, 1	M1	Valid	Valid	Unsigned divide r/m16 by 2, once.
D3 /5	SHR r/m16, CL	MC	Valid	Valid	Unsigned divide r/m16 by 2, CL times
C1 /5 ib	SHR r/m16, imm8	MI	Valid	Valid	Unsigned divide r/m16 by 2, imm8 times.
D1 /5	SHR r/m32, 1	M1	Valid	Valid	Unsigned divide r/m32 by 2, once.
REX.W + D1 /5	SHR r/m64, 1	M1	Valid	N.E.	Unsigned divide r/m64 by 2, once.
D3 /5	SHR r/m32, CL	MC	Valid	Valid	Unsigned divide r/m32 by 2, CL times.
REX.W + D3 /5	SHR r/m64, CL	MC	Valid	N.E.	Unsigned divide r/m64 by 2, CL times.
C1 /5 ib	SHR r/m32, imm8	MI	Valid	Valid	Unsigned divide r/m32 by 2, imm8 times.
REX.W + C1 /5 ib	SHR r/m64, imm8	MI	Valid	N.E.	Unsigned divide r/m64 by 2, imm8 times.

**NOTES:**

\* Not the same form of division as IDIV; rounding is toward negative infinity.

\*\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

\*\*\*See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (r, w)	1	NA	NA
MC	ModRM:r/m (r, w)	CL	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

**Description**

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W is used). The count range is limited to 0 to 31 (or 63 if 64-bit mode and REX.W is used). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the “quotient” of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the “remainder” is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction’s default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

## IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

## Operation

IF 64-Bit Mode and using REX.W

```

    THEN
        countMASK ← 3FH;
    ELSE
        countMASK ← 1FH;
    FI

    tempCOUNT ← (COUNT AND countMASK);
    tempDEST ← DEST;
    WHILE (tempCOUNT ≠ 0)
    DO
        IF instruction is SAL or SHL
            THEN
                CF ← MSB(DEST);
            ELSE (* Instruction is SAR or SHR *)
                CF ← LSB(DEST);
            FI;
        IF instruction is SAL or SHL
            THEN
                DEST ← DEST * 2;
            ELSE
                IF instruction is SAR

```

```

        THEN
            DEST ← DEST / 2; (* Signed divide, rounding toward negative infinity *)
        ELSE (* Instruction is SHR *)
            DEST ← DEST / 2 ; (* Unsigned divide *)
        Fl;
    Fi;
    tempCOUNT ← tempCOUNT - 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* Instruction is SHR *)
                        OF ← MSB(tempDEST);
                    Fl;
                Fi;
        ELSE IF (COUNT AND countMASK) = 0
            THEN
                All flags unchanged;
            ELSE (* COUNT not 1 or 0 *)
                OF ← undefined;
        Fl;
    Fi;

```

### Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**SARX/SHLX/SHRX – Shift Without Affecting Flags**

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS <sup>1</sup> .LZ.F3.0F38.W0 F7 /r SARX r32a, r/m32, r32b	RMV	V/V	BMI2	Shift r/m32 arithmetically right with count specified in r32b.
VEX.NDS <sup>1</sup> .LZ.66.0F38.W0 F7 /r SHLX r32a, r/m32, r32b	RMV	V/V	BMI2	Shift r/m32 logically left with count specified in r32b.
VEX.NDS <sup>1</sup> .LZ.F2.0F38.W0 F7 /r SHRX r32a, r/m32, r32b	RMV	V/V	BMI2	Shift r/m32 logically right with count specified in r32b.
VEX.NDS <sup>1</sup> .LZ.F3.0F38.W1 F7 /r SARX r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Shift r/m64 arithmetically right with count specified in r64b.
VEX.NDS <sup>1</sup> .LZ.66.0F38.W1 F7 /r SHLX r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Shift r/m64 logically left with count specified in r64b.
VEX.NDS <sup>1</sup> .LZ.F2.0F38.W1 F7 /r SHRX r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Shift r/m64 logically right with count specified in r64b.

**NOTES:**

1. ModRM:r/m is used to encode the first source operand (second operand) and VEX.vvvv encodes the second source operand (third operand).

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

**Description**

Shifts the bits of the first source operand (the second operand) to the left or right by a COUNT value specified in the second source operand (the third operand). The result is written to the destination operand (the first operand).

The shift arithmetic right (SARX) and shift logical right (SHRX) instructions shift the bits of the destination operand to the right (toward less significant bit locations), SARX keeps and propagates the most significant bit (sign bit) while shifting.

The logical shift left (SHLX) shifts the bits of the destination operand to the left (toward more significant bit locations).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

If the value specified in the first source operand exceeds OperandSize - 1, the COUNT value is masked.

SARX, SHRX, and SHLX instructions do not update flags.

**Operation**

```
TEMP ← SRC1;
IF VEX.W1 and CS.L = 1
THEN
    countMASK ← 3FH;
```

```

ELSE
    countMASK ← 1FH;
FI
COUNT ← (SRC2 AND countMASK)

DEST[OperandSize - 1] = TEMP[OperandSize - 1];
DO WHILE (COUNT != 0)
    IF instruction is SHLX
        THEN
            DEST[] ← DEST *2;
        ELSE IF instruction is SHRX
            THEN
                DEST[] ← DEST /2; //unsigned divide
            ELSE      // SARX
                DEST[] ← DEST /2; // signed divide, round toward negative infinity
            FI;
        COUNT ← COUNT - 1;
OD

```

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
#UD              If VEX.W = 1.

## SBB—Integer Subtraction with Borrow

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
1C <i>ib</i>	SBB AL, <i>imm8</i>	I	Valid	Valid	Subtract with borrow <i>imm8</i> from AL.
1D <i>iw</i>	SBB AX, <i>imm16</i>	I	Valid	Valid	Subtract with borrow <i>imm16</i> from AX.
1D <i>id</i>	SBB EAX, <i>imm32</i>	I	Valid	Valid	Subtract with borrow <i>imm32</i> from EAX.
REX.W + 1D <i>id</i>	SBB RAX, <i>imm32</i>	I	Valid	N.E.	Subtract with borrow sign-extended <i>imm32</i> to 64-bits from RAX.
80 /3 <i>ib</i>	SBB r/m8, <i>imm8</i>	MI	Valid	Valid	Subtract with borrow <i>imm8</i> from r/m8.
REX + 80 /3 <i>ib</i>	SBB r/m8*, <i>imm8</i>	MI	Valid	N.E.	Subtract with borrow <i>imm8</i> from r/m8.
81 /3 <i>iw</i>	SBB r/m16, <i>imm16</i>	MI	Valid	Valid	Subtract with borrow <i>imm16</i> from r/m16.
81 /3 <i>id</i>	SBB r/m32, <i>imm32</i>	MI	Valid	Valid	Subtract with borrow <i>imm32</i> from r/m32.
REX.W + 81 /3 <i>id</i>	SBB r/m64, <i>imm32</i>	MI	Valid	N.E.	Subtract with borrow sign-extended <i>imm32</i> to 64-bits from r/m64.
83 /3 <i>ib</i>	SBB r/m16, <i>imm8</i>	MI	Valid	Valid	Subtract with borrow sign-extended <i>imm8</i> from r/m16.
83 /3 <i>ib</i>	SBB r/m32, <i>imm8</i>	MI	Valid	Valid	Subtract with borrow sign-extended <i>imm8</i> from r/m32.
REX.W + 83 /3 <i>ib</i>	SBB r/m64, <i>imm8</i>	MI	Valid	N.E.	Subtract with borrow sign-extended <i>imm8</i> from r/m64.
18 /r	SBB r/m8, r8	MR	Valid	Valid	Subtract with borrow r8 from r/m8.
REX + 18 /r	SBB r/m8*, r8	MR	Valid	N.E.	Subtract with borrow r8 from r/m8.
19 /r	SBB r/m16, r16	MR	Valid	Valid	Subtract with borrow r16 from r/m16.
19 /r	SBB r/m32, r32	MR	Valid	Valid	Subtract with borrow r32 from r/m32.
REX.W + 19 /r	SBB r/m64, r64	MR	Valid	N.E.	Subtract with borrow r64 from r/m64.
1A /r	SBB r8, r/m8	RM	Valid	Valid	Subtract with borrow r/m8 from r8.
REX + 1A /r	SBB r8*, r/m8*	RM	Valid	N.E.	Subtract with borrow r/m8 from r8.
1B /r	SBB r16, r/m16	RM	Valid	Valid	Subtract with borrow r/m16 from r16.
1B /r	SBB r32, r/m32	RM	Valid	Valid	Subtract with borrow r/m32 from r32.
REX.W + 1B /r	SBB r64, r/m64	RM	Valid	N.E.	Subtract with borrow r/m64 from r64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (w)	imm8/16/32	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

## Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

$\text{DEST} \leftarrow (\text{DEST} - (\text{SRC} + \text{CF}))$ ;

## 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 <i>m8</i>	NP	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags.*
AF	SCAS <i>m16</i>	NP	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags.*
AF	SCAS <i>m32</i>	NP	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCAS <i>m64</i>	NP	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.
AE	SCASB	NP	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI then set status flags.*
AF	SCASW	NP	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags.*
AF	SCASD	NP	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCASQ	NP	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.

### NOTES:

\* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES: (E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed. The explicit-operand form and the no-operands form. The explicit-operand form (specified using the SCAS mnemonic) allows a memory operand to be specified explicitly. The memory operand must be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (AL register for byte comparisons, AX for word comparisons, EAX for doubleword comparisons). The explicit-operand form is provided to allow documentation. Note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword) but it does not have to specify the correct location. The location is always specified by ES: (E)DI.

The no-operands form of the instruction uses a short form of SCAS. Again, ES: (E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes

some action based on the setting of status flags. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

## Operation

Non-64-bit Mode:

```
IF (Byte comparison)
  THEN
    temp ← AL – SRC;
    SetStatusFlags(temp);
    THEN IF DF = 0
      THEN (E)DI ← (E)DI + 1;
      ELSE (E)DI ← (E)DI - 1; Fl;
  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; Fl;
    Fl;
  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; Fl;
    Fl;
Fl;
```

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; Fl;
  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; Fl;
    Fl;
```

```

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; Fl;
    Fl;
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;
    Fl;
Fl;
F

```

### Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment. If the ES register contains a NULL segment selector. If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

## INSTRUCTION SET REFERENCE, N-Z

- #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.

## SETcc—Set Byte on Condition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA r/m8	M	Valid	Valid	Set byte if above (CF=0 and ZF=0).
REX + 0F 97	SETA r/m8*	M	Valid	N.E.	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE r/m8	M	Valid	Valid	Set byte if above or equal (CF=0).
REX + 0F 93	SETAE r/m8*	M	Valid	N.E.	Set byte if above or equal (CF=0).
0F 92	SETB r/m8	M	Valid	Valid	Set byte if below (CF=1).
REX + 0F 92	SETB r/m8*	M	Valid	N.E.	Set byte if below (CF=1).
0F 96	SETBE r/m8	M	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
REX + 0F 96	SETBE r/m8*	M	Valid	N.E.	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC r/m8	M	Valid	Valid	Set byte if carry (CF=1).
REX + 0F 92	SETC r/m8*	M	Valid	N.E.	Set byte if carry (CF=1).
0F 94	SETE r/m8	M	Valid	Valid	Set byte if equal (ZF=1).
REX + 0F 94	SETE r/m8*	M	Valid	N.E.	Set byte if equal (ZF=1).
0F 9F	SETG r/m8	M	Valid	Valid	Set byte if greater (ZF=0 and SF=OF).
REX + 0F 9F	SETG r/m8*	M	Valid	N.E.	Set byte if greater (ZF=0 and SF=OF).
0F 9D	SETGE r/m8	M	Valid	Valid	Set byte if greater or equal (SF=OF).
REX + 0F 9D	SETGE r/m8*	M	Valid	N.E.	Set byte if greater or equal (SF=OF).
0F 9C	SETL r/m8	M	Valid	Valid	Set byte if less (SF≠ OF).
REX + 0F 9C	SETL r/m8*	M	Valid	N.E.	Set byte if less (SF≠ OF).
0F 9E	SETLE r/m8	M	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ OF).
REX + 0F 9E	SETLE r/m8*	M	Valid	N.E.	Set byte if less or equal (ZF=1 or SF≠ OF).
0F 96	SETNA r/m8	M	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
REX + 0F 96	SETNA r/m8*	M	Valid	N.E.	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE r/m8	M	Valid	Valid	Set byte if not above or equal (CF=1).
REX + 0F 92	SETNAE r/m8*	M	Valid	N.E.	Set byte if not above or equal (CF=1).
0F 93	SETNB r/m8	M	Valid	Valid	Set byte if not below (CF=0).
REX + 0F 93	SETNB r/m8*	M	Valid	N.E.	Set byte if not below (CF=0).
0F 97	SETNBE r/m8	M	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).
REX + 0F 97	SETNBE r/m8*	M	Valid	N.E.	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC r/m8	M	Valid	Valid	Set byte if not carry (CF=0).
REX + 0F 93	SETNC r/m8*	M	Valid	N.E.	Set byte if not carry (CF=0).
0F 95	SETNE r/m8	M	Valid	Valid	Set byte if not equal (ZF=0).
REX + 0F 95	SETNE r/m8*	M	Valid	N.E.	Set byte if not equal (ZF=0).
0F 9E	SETNG r/m8	M	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ OF).
REX + 0F 9E	SETNG r/m8*	M	Valid	N.E.	Set byte if not greater (ZF=1 or SF≠ OF).
0F 9C	SETNGE r/m8	M	Valid	Valid	Set byte if not greater or equal (SF≠ OF).
REX + 0F 9C	SETNGE r/m8*	M	Valid	N.E.	Set byte if not greater or equal (SF≠ OF).
0F 9D	SETNL r/m8	M	Valid	Valid	Set byte if not less (SF=OF).
REX + 0F 9D	SETNL r/m8*	M	Valid	N.E.	Set byte if not less (SF=OF).
0F 9F	SETNLE r/m8	M	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=OF).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX + OF 9F	SETNLE r/m8*	M	Valid	N.E.	Set byte if not less or equal (ZF=0 and SF=OF).
OF 91	SETNO r/m8	M	Valid	Valid	Set byte if not overflow (OF=0).
REX + OF 91	SETNO r/m8*	M	Valid	N.E.	Set byte if not overflow (OF=0).
OF 9B	SETNP r/m8	M	Valid	Valid	Set byte if not parity (PF=0).
REX + OF 9B	SETNP r/m8*	M	Valid	N.E.	Set byte if not parity (PF=0).
OF 99	SETNS r/m8	M	Valid	Valid	Set byte if not sign (SF=0).
REX + OF 99	SETNS r/m8*	M	Valid	N.E.	Set byte if not sign (SF=0).
OF 95	SETNZ r/m8	M	Valid	Valid	Set byte if not zero (ZF=0).
REX + OF 95	SETNZ r/m8*	M	Valid	N.E.	Set byte if not zero (ZF=0).
OF 90	SETO r/m8	M	Valid	Valid	Set byte if overflow (OF=1)
REX + OF 90	SETO r/m8*	M	Valid	N.E.	Set byte if overflow (OF=1).
OF 9A	SETP r/m8	M	Valid	Valid	Set byte if parity (PF=1).
REX + OF 9A	SETP r/m8*	M	Valid	N.E.	Set byte if parity (PF=1).
OF 9A	SETPE r/m8	M	Valid	Valid	Set byte if parity even (PF=1).
REX + OF 9A	SETPE r/m8*	M	Valid	N.E.	Set byte if parity even (PF=1).
OF 9B	SETPO r/m8	M	Valid	Valid	Set byte if parity odd (PF=0).
REX + OF 9B	SETPO r/m8*	M	Valid	N.E.	Set byte if parity odd (PF=0).
OF 98	SETS r/m8	M	Valid	Valid	Set byte if sign (SF=1).
REX + OF 98	SETS r/m8*	M	Valid	N.E.	Set byte if sign (SF=1).
OF 94	SETZ r/m8	M	Valid	Valid	Set byte if zero (ZF=1).
REX + OF 94	SETZ r/m8*	M	Valid	N.E.	Set byte if zero (ZF=1).

**NOTES:**

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

**Description**

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms “above” and “below” are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms “greater” and “less” are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SET*cc* instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, “EFLAGS Condition Codes,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SET*cc* instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction's operation is the same as in legacy mode and compatibility mode.

## Operation

If condition

```
THEN DEST ← 1;  
ELSE DEST ← 0;  
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

**SFENCE—Store Fence**

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /7	SFENCE	NP	Valid	Valid	Serializes store operations.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

**Description**

Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes the SFENCE instruction in program order becomes globally visible before any store instruction that follows the SFENCE instruction. The SFENCE instruction is ordered with respect to store instructions, other SFENCE instructions, any LFENCE and MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

```
Wait_On_Following_Stores_Until(preceding_stores_globally_visible);
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
void _mm_sfence(void)
```

**Exceptions (All Operating Modes)**

#UD                    If CPUID.01H:EDX.SSE[bit 25] = 0.  
                       If the LOCK prefix is used.

## SGDT—Store Global Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /0	SGDT <i>m</i>	M	Valid	Valid	Store GDTR to <i>m</i> .

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in bytes 3-5, and byte 6 is zero-filled. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In IA-32e mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

### IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 processor family, Pentium, Intel486, and Intel386™ processors fill these bits with 0s.

### Operation

IF instruction is SGDT

```

IF OperandSize = 16
  THEN
    DEST[0:15] ← GDTR(Limit);
    DEST[16:39] ← GDTR(Base); (* 24 bits of base address stored *)
    DEST[40:47] ← 0;
  ELSE IF (32-bit Operand Size)
    DEST[0:15] ← GDTR(Limit);
    DEST[16:47] ← GDTR(Base); (* Full 32-bit base address stored *)
    FI;
  ELSE (* 64-bit Operand Size *)
    DEST[0:15] ← GDTR(Limit);
    DEST[16:79] ← GDTR(Base); (* Full 64-bit base address stored *)
    FI;
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#UD	If the destination operand is a register. If the LOCK prefix is used.
#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#UD	If the destination operand is a register. If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#UD	If the destination operand is a register. If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the destination operand is a register. If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SHLD—Double Precision Shift Left

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF A4 /r ib	SHLD <i>r/m16, r16, imm8</i>	MRI	Valid	Valid	Shift <i>r/m16</i> to left <i>imm8</i> places while shifting bits from <i>r16</i> in from the right.
OF A5 /r	SHLD <i>r/m16, r16, CL</i>	MRC	Valid	Valid	Shift <i>r/m16</i> to left CL places while shifting bits from <i>r16</i> in from the right.
OF A4 /r ib	SHLD <i>r/m32, r32, imm8</i>	MRI	Valid	Valid	Shift <i>r/m32</i> to left <i>imm8</i> places while shifting bits from <i>r32</i> in from the right.
REX.W + OF A4 /r ib	SHLD <i>r/m64, r64, imm8</i>	MRI	Valid	N.E.	Shift <i>r/m64</i> to left <i>imm8</i> places while shifting bits from <i>r64</i> in from the right.
OF A5 /r	SHLD <i>r/m32, r32, CL</i>	MRC	Valid	Valid	Shift <i>r/m32</i> to left CL places while shifting bits from <i>r32</i> in from the right.
REX.W + OF A5 /r	SHLD <i>r/m64, r64, CL</i>	MRC	Valid	N.E.	Shift <i>r/m64</i> to left CL places while shifting bits from <i>r64</i> in from the right.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

### Description

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE

```

```

IF COUNT > SIZE
    THEN (* Bad parameters *)
        DEST is undefined;
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
        CF ← BIT[DEST, SIZE - COUNT];
        (* Last bit shifted out on exit *)
        FOR i ← SIZE - 1 DOWN TO COUNT
            DO
                Bit(DEST, i) ← Bit(DEST, i - COUNT);
            OD;
        FOR i ← COUNT - 1 DOWN TO 0
            DO
                BIT[DEST, i] ← BIT[SRC, i - COUNT + SIZE];
            OD;
        FI;
    FI;

```

### Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**SHRD—Double Precision Shift Right**

<b>Opcode*</b>	<b>Instruction</b>	<b>Op/ En</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
OF AC /r ib	SHRD <i>r/m16, r16, imm8</i>	MRI	Valid	Valid	Shift <i>r/m16</i> to right <i>imm8</i> places while shifting bits from <i>r16</i> in from the left.
OF AD /r	SHRD <i>r/m16, r16, CL</i>	MRC	Valid	Valid	Shift <i>r/m16</i> to right CL places while shifting bits from <i>r16</i> in from the left.
OF AC /r ib	SHRD <i>r/m32, r32, imm8</i>	MRI	Valid	Valid	Shift <i>r/m32</i> to right <i>imm8</i> places while shifting bits from <i>r32</i> in from the left.
REX.W + OF AC /r ib	SHRD <i>r/m64, r64, imm8</i>	MRI	Valid	N.E.	Shift <i>r/m64</i> to right <i>imm8</i> places while shifting bits from <i>r64</i> in from the left.
OF AD /r	SHRD <i>r/m32, r32, CL</i>	MRC	Valid	Valid	Shift <i>r/m32</i> to right CL places while shifting bits from <i>r32</i> in from the left.
REX.W + OF AD /r	SHRD <i>r/m64, r64, CL</i>	MRC	Valid	N.E.	Shift <i>r/m64</i> to right CL places while shifting bits from <i>r64</i> in from the left.

**Instruction Operand Encoding**

<b>Op/En</b>	<b>Operand 1</b>	<b>Operand 2</b>	<b>Operand 3</b>	<b>Operand 4</b>
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

**Description**

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE

```

```

IF COUNT > SIZE
    THEN (* Bad parameters *)
        DEST is undefined;
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
        CF ← BIT[DEST, COUNT - 1]; (* Last bit shifted out on exit *)
        FOR i ← 0 TO SIZE - 1 - COUNT
            DO
                BIT[DEST, i] ← BIT[DEST, i + COUNT];
            OD;
        FOR i ← SIZE - COUNT TO SIZE - 1
            DO
                BIT[DEST,i] ← BIT[SRC, i + COUNT - SIZE];
            OD;
        FI;
    FI;

```

### Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

## SHUFPD—Shuffle Packed Double-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle packed double-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by <i>imm8</i> from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by <i>imm8</i> from <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Moves either of the two packed double-precision floating-point values from destination operand (first operand) into the low quadword of the destination operand; moves either of the two packed double-precision floating-point values from the source operand into the high quadword of the destination operand (see Figure 4-21). The select operand (third operand) determines which values are moved to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

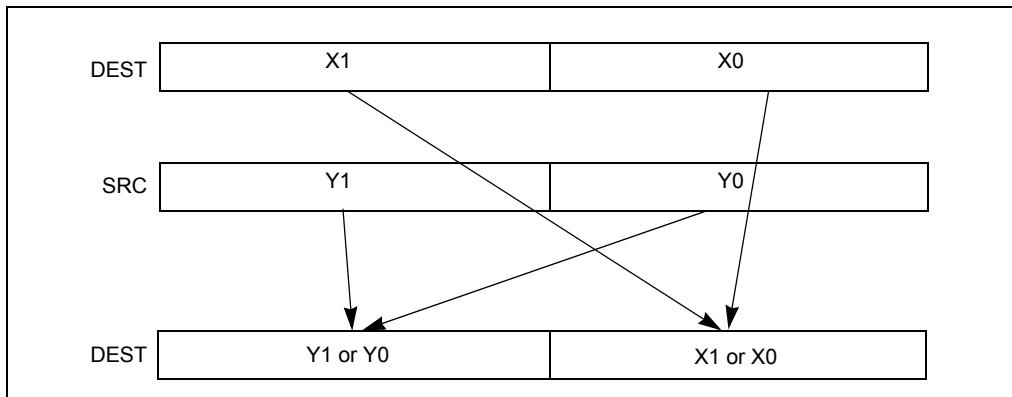


Figure 4-21. SHUFPD Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bit 0 selects which value is moved from the destination operand to the result (where 0 selects the low quadword and 1 selects the high quadword) and bit 1 selects which value is moved from the source operand to the result. Bits 2 through 7 of the select operand are reserved and must be set to 0.

## Operation

```
IF SELECT[0] = 0
    THEN DEST[63:0] ← DEST[63:0];
ELSE DEST[63:0] ← DEST[127:64]; Fl;
```

```
IF SELECT[1] = 0
    THEN DEST[127:64] ← SRC[63:0];
ELSE DEST[127:64] ← SRC[127:64]; Fl;
```

### SHUFPD (128-bit Legacy SSE version)

```
IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] Fl;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] Fl;
DEST[VLMAX-1:128] (Unmodified)
```

### VSHUFPD (VEX.128 encoded version)

```
IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] Fl;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] Fl;
DEST[VLMAX-1:128] ← 0
```

### VSHUFPD (VEX.256 encoded version)

```
IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] Fl;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] Fl;
IF IMMO[2] = 0
    THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST[191:128] ← SRC1[255:192] Fl;
IF IMMO[3] = 0
    THEN DEST[255:192] ← SRC2[191:128]
    ELSE DEST[255:192] ← SRC2[255:192] Fl;
```

## Intel C/C++ Compiler Intrinsic Equivalent

SHUFPD:	<code>__m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)</code>
VSHUFPD:	<code>__m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);</code>

## SIMD Floating-Point Exceptions

None.

**Other Exceptions**

See Exceptions Type 4.

## SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C6 /r ib SHUFPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE	Shuffle packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm1/m128</i> to <i>xmm1</i> .
VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by <i>imm8</i> from <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Moves two of the four packed single-precision floating-point values from the destination operand (first operand) into the low quadword of the destination operand; moves two of the four packed single-precision floating-point values from the source operand (second operand) into to the high quadword of the destination operand (see Figure 4-22). The select operand (third operand) determines which values are moved to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

determines which values are moved to the destination operand.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

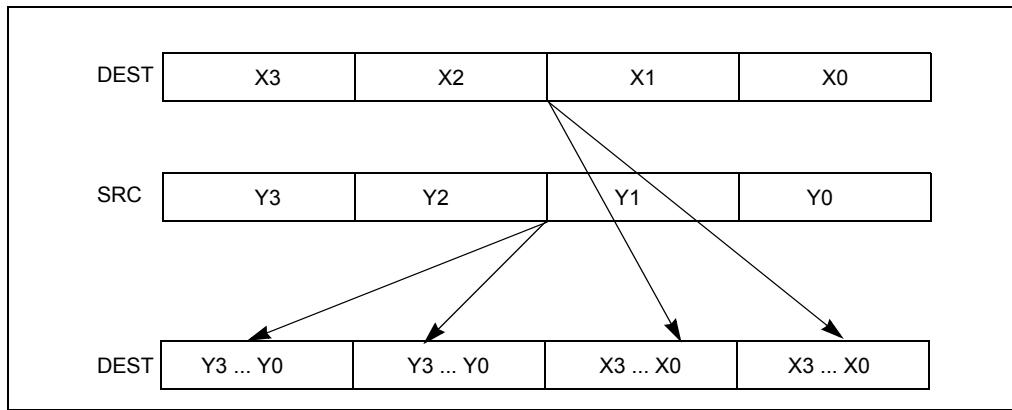


Figure 4-22. SHUFPS Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bits 0 and 1 select the value to be moved from the destination operand to the low doubleword of the result, bits 2 and 3 select the value to be moved from the destination operand to the second doubleword of the result, bits 4 and 5 select the value to be moved from the source operand to the third doubleword of the result, and bits 6 and 7 select the value to be moved from the source operand to the high doubleword of the result.

## Operation

CASE (SELECT[1:0]) OF

```
0: DEST[31:0] ← DEST[31:0];
1: DEST[31:0] ← DEST[63:32];
2: DEST[31:0] ← DEST[95:64];
3: DEST[31:0] ← DEST[127:96];
```

ESAC;

CASE (SELECT[3:2]) OF

```
0: DEST[63:32] ← DEST[31:0];
1: DEST[63:32] ← DEST[63:32];
2: DEST[63:32] ← DEST[95:64];
3: DEST[63:32] ← DEST[127:96];
```

ESAC;

CASE (SELECT[5:4]) OF

```
0: DEST[95:64] ← SRC[31:0];
1: DEST[95:64] ← SRC[63:32];
2: DEST[95:64] ← SRC[95:64];
3: DEST[95:64] ← SRC[127:96];
```

ESAC;

CASE (SELECT[7:6]) OF

```
0: DEST[127:96] ← SRC[31:0];
1: DEST[127:96] ← SRC[63:32];
2: DEST[127:96] ← SRC[95:64];
3: DEST[127:96] ← SRC[127:96];
```

ESAC;

## SHUFPS (128-bit Legacy SSE version)

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);  
 DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);  
 DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);  
 DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);  
 DEST[VLMAX-1:128] (Unmodified)

## VSHUFPS (VEX.128 encoded version)

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);  
 DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);  
 DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);  
 DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);  
 DEST[VLMAX-1:128] ← 0

**VSHUFPS (VEX.256 encoded version)**

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

SHUFPS:    __m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)
VSHUFPS:   __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4.

## SIDT—Store Interrupt Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 /1	SIDT <i>m</i>	M	Valid	Valid	Store IDTR to <i>m</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, if the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with Os.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

### IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 processor family, Pentium, Intel486, and Intel386 processors fill these bits with Os.

### Operation

IF instruction is SIDT

THEN

  IF OperandSize = 16

    THEN

      DEST[0:15] ← IDTR(Limit);

      DEST[16:39] ← IDTR(Base); (\* 24 bits of base address stored; \*)

      DEST[40:47] ← 0;

    ELSE IF (32-bit Operand Size)

      DEST[0:15] ← IDTR(Limit);

      DEST[16:47] ← IDTR(Base); Fl; (\* Full 32-bit base address stored \*)

    ELSE (\* 64-bit Operand Size \*)

      DEST[0:15] ← IDTR(Limit);

      DEST[16:79] ← IDTR(Base); (\* Full 64-bit base address stored \*)

  Fl;

Fl;

### Flags Affected

None.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the destination operand is a register.
	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SLDT—Store Local Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 00 /0	SLDT r/m16	M	Valid	Valid	Stores segment selector from LDTR in r/m16.
REX.W + OF 00 /0	SLDT r64/m16	M	Valid	Valid	Stores segment selector from LDTR in r64/m16.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

### Operation

DEST ← LDTR(SegmentSelector);

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the destination is located in a non-writable segment.  
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

- #UD The SLDT instruction is not recognized in real-address mode.

If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

- #UD                    The SLDT instruction is not recognized in virtual-8086 mode.  
                      If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #SS(0)                If a memory address referencing the SS segment is in a non-canonical form.  
#GP(0)                If the memory address is in a non-canonical form.  
#PF(fault-code)      If a page fault occurs.  
#AC(0)                If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.  
#UD                    If the LOCK prefix is used.

## SMSW—Store Machine Status Word

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 /4	SMSW <i>r/m16</i>	M	Valid	Valid	Store machine status word to <i>r/m16</i> .
OF 01 /4	SMSW <i>r32/m16</i>	M	Valid	Valid	Store machine status word in low-order 16 bits of <i>r32/m16</i> ; high-order 16 bits of <i>r32</i> are undefined.
REX.W + OF 01 /4	SMSW <i>r64/m16</i>	M	Valid	Valid	Store machine status word in low-order 16 bits of <i>r64/m16</i> ; high-order 16 bits of <i>r32</i> are undefined.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the machine status word (bits 0 through 15 of control register CRO) 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 CRO 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 CRO 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 CRO[15:0] in r16
- SMSW r32 operand size 32, zero-extend CRO[31:0], and store in r32
- SMSW r64 operand size 64, zero-extend CRO[63:0], and store in r64
- SMSW m16 operand size 16, store CRO[15:0] in m16
- SMSW m16 operand size 32, store CRO[15:0] in m16 (not m32)
- SMSW m16 operands size 64, store CRO[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. This is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the machine status word.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

DEST  $\leftarrow$  CRO[15:0];  
(\* Machine status word \*)

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 51 /r SQRTPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Computes square roots of the packed double-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.66.0F.WIG 51 /r VSQRTPD <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in <i>xmm2/m128</i> and stores the result in <i>xmm1</i> .
VEX.256.66.0F.WIG 51/r VSQRTPD <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in <i>ymm2/m256</i> and stores the result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the square roots of the two packed double-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### SQRTPD (128-bit Legacy SSE version)

DEST[63:0]  $\leftarrow$  SQRT(SRC[63:0])  
 DEST[127:64]  $\leftarrow$  SQRT(SRC[127:64])  
 DEST[VLMAX-1:128] (Unmodified)

#### VSQRTPD (VEX.128 encoded version)

DEST[63:0]  $\leftarrow$  SQRT(SRC[63:0])  
 DEST[127:64]  $\leftarrow$  SQRT(SRC[127:64])  
 DEST[VLMAX-1:128]  $\leftarrow$  0

**VSQRTPD (VEX.256 encoded version)**

DEST[63:0]  $\leftarrow$  SQRT(SRC[63:0])  
DEST[127:64]  $\leftarrow$  SQRT(SRC[127:64])  
DEST[191:128]  $\leftarrow$  SQRT(SRC[191:128])  
DEST[255:192]  $\leftarrow$  SQRT(SRC[255:192])

**Intel C/C++ Compiler Intrinsic Equivalent**

SQRTPD:       `__m128d _mm_sqrt_pd (__m128d a)`  
SQRTPD:       `__m256d _mm256_sqrt_pd (__m256d a);`

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2; additionally

#UD              If VEX.vvvv != 1111B.

## SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 51 /r SQRTPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Computes square roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.0F.WIG 51 /r VSQRTPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the result in <i>xmm1</i> .
VEX.256.0F.WIG 51/r VSQRTPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in <i>ymm2/m256</i> and stores the result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the square roots of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### SQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SQRT(SRC[31:0])
DEST[63:32] ← SQRT(SRC[63:32])
DEST[95:64] ← SQRT(SRC[95:64])
DEST[127:96] ← SQRT(SRC[127:96])
DEST[VLMAX-1:128] (Unmodified)
```

#### VSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ← SQRT(SRC[31:0])
DEST[63:32] ← SQRT(SRC[63:32])
DEST[95:64] ← SQRT(SRC[95:64])
DEST[127:96] ← SQRT(SRC[127:96])
DEST[VLMAX-1:128] ← 0
```

**VSQRTPS (VEX.256 encoded version)**

```

DEST[31:0] ← SQRT(SRC[31:0])
DEST[63:32] ← SQRT(SRC[63:32])
DEST[95:64] ← SQRT(SRC[95:64])
DEST[127:96] ← SQRT(SRC[127:96])
DEST[159:128] ← SQRT(SRC[159:128])
DEST[191:160] ← SQRT(SRC[191:160])
DEST[223:192] ← SQRT(SRC[223:192])
DEST[255:224] ← SQRT(SRC[255:224])

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

SQRTPS:      _m128 _mm_sqrt_ps(_m128 a)
SQRTPS:      _m256 _mm256_sqrt_ps (_m256 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2; additionally

#UD                  If VEX.vvvv != 1111B.

## SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 OF 51 /r SQRTSD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE2	Computes square root of the low double-precision floating-point value in <i>xmm2/m64</i> and stores the results in <i>xmm1</i> .
VEX.NDS.LIG.F2.OF.WIG 51/r VSQRTSD <i>xmm1,xmm2, xmm3/m64</i>	RVM	V/V	AVX	Computes square root of the low double-precision floating point value in <i>xmm3/m64</i> and stores the results in <i>xmm2</i> . Also, upper double precision floating-point value (bits[127:64]) from <i>xmm2</i> are copied to <i>xmm1[127:64]</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the square root of the low double-precision floating-point value in the source operand (second operand) and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

#### SQRTSD (128-bit Legacy SSE version)

DEST[63:0]  $\leftarrow$  SQRT(SRC[63:0])  
DEST[VLMAX-1:64] (Unmodified)

#### VSQRTSD (VEX.128 encoded version)

DEST[63:0]  $\leftarrow$  SQRT(SRC2[63:0])  
DEST[127:64]  $\leftarrow$  SRC1[127:64]  
DEST[VLMAX-1:128]  $\leftarrow$  0

### Intel C/C++ Compiler Intrinsic Equivalent

SQRTSD: `__m128d _mm_sqrt_sd (m128d a, m128d b)`

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

### Other Exceptions

See Exceptions Type 3.

**SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value**

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 51 /r SQRTSS <i>xmm1, xmm2/m32</i>	RM	V/V	SSE	Computes square root of the low single-precision floating-point value in <i>xmm2/m32</i> and stores the results in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 51/r VSQRTSS <i>xmm1, xmm2, xmm3/m32</i>	RVM	V/V	AVX	Computes square root of the low single-precision floating-point value in <i>xmm3/m32</i> and stores the results in <i>xmm1</i> . Also, upper single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1[127:32]</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Computes the square root of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order double-words of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation****SQRTSS (128-bit Legacy SSE version)**

DEST[31:0]  $\leftarrow$  SQRT(SRC2[31:0])  
DEST[VLMAX-1:32] (Unmodified)

**VSQRTSS (VEX.128 encoded version)**

DEST[31:0]  $\leftarrow$  SQRT(SRC2[31:0])  
DEST[127:32]  $\leftarrow$  SRC1[127:32]  
DEST[VLMAX-1:128]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

SQRTSS: `__m128_mm_sqrt_ss(__m128 a)`

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 3.

**STC—Set Carry Flag**

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F9	STC	NP	Valid	Valid	Set CF flag.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

**Description**

Sets the CF flag in the EFLAGS register.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

$CF \leftarrow 1;$

**Flags Affected**

The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD                  If the LOCK prefix is used.

**STD—Set Direction Flag**

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FD	STD	NP	Valid	Valid	Set DF flag.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

**Description**

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

$DF \leftarrow 1;$

**Flags Affected**

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD                  If the LOCK prefix is used.

## STI—Set Interrupt Flag

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FB	STI	NP	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

If protected-mode virtual interrupts are not enabled, STI sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized<sup>1</sup>. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and NMI interrupts. NMI interrupts (and SMIs) may be blocked for one macroinstruction following an STI.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-15 indicates the action of the STI instruction depending on the processor's mode of operation and the CPL/IOPL settings of the running program or procedure.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Table 4-15. Decision Table for STI Results**

PE	VM	IOPL	CPL	PVI	VIP	VME	STI Result
0	X	X	X	X	X	X	IF = 1
1	0	$\geq$ CPL	X	X	X	X	IF = 1
1	0	< CPL	3	1	0	X	VIF = 1
1	0	< CPL	< 3	X	X	X	GP Fault
1	0	< CPL	X	0	X	X	GP Fault
1	0	< CPL	X	X	1	X	GP Fault
1	1	3	X	X	X	X	IF = 1
1	1	< 3	X	X	0	1	VIF = 1
1	1	< 3	X	X	1	X	GP Fault
1	1	< 3	X	X	X	0	GP Fault

### NOTES:

X = This setting has no impact.

1. The STI instruction delays recognition of interrupts only if it is executed with EFLAGS.IF = 0. In a sequence of STI instructions, only the first instruction in the sequence is guaranteed to delay interrupts.

In the following instruction sequence, interrupts may be recognized before RET executes:

STI  
STI  
RET

## Operation

```

IF PE = 0 (* Executing in real-address mode *)
  THEN
    IF ← 1; (* Set Interrupt Flag *)
  ELSE (* Executing in protected mode or virtual-8086 mode *)
    IF VM = 0 (* Executing in protected mode*)
      THEN
        IF IOPL ≥ CPL
          THEN
            IF ← 1; (* Set Interrupt Flag *)
          ELSE
            IF (IOPL < CPL) and (CPL = 3) and (VIP = 0)
              THEN
                VIF ← 1; (* Set Virtual Interrupt Flag *)
              ELSE
                #GP(0);
              FI;
            FI;
          FI;
        ELSE (* Executing in Virtual-8086 mode *)
          IF IOPL = 3
            THEN
              IF ← 1; (* Set Interrupt Flag *)
            ELSE
              IF ((IOPL < 3) and (VIP = 0) and (VME = 1))
                THEN
                  VIF ← 1; (* Set Virtual Interrupt Flag *)
                ELSE
                  #GP(0); (* Trap to virtual-8086 monitor *)
                FI;
              FI;
            FI;
          FI;
        FI;
      FI;
    FI;
  FI;

```

## Flags Affected

The IF flag is set to 1; or the VIF flag is set to 1.

## Protected Mode Exceptions

- #GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.
- #UD If the LOCK prefix is used.

## Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF AE /3 STMXCSR <i>m32</i>	M	V/V	SSE	Store contents of MXCSR register to <i>m32</i> .
VEX.LZ.OF.WIG AE /3 VSTMCSR <i>m32</i>	M	V/V	AVX	Store contents of MXCSR register to <i>m32</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

VEX.L must be 0, otherwise instructions will #UD.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

*m32* ← MXCSR;

### Intel C/C++ Compiler Intrinsic Equivalent

\_mm\_getcsr(void)

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 5; additionally

#UD                    If VEX.L= 1,  
                       If VEX.vvvv != 1111B.

**STOS/STOSB/STOSW/STOSD/STOSQ—Store String**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AA	STOS <i>m8</i>	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOS <i>m16</i>	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOS <i>m32</i>	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOS <i>m64</i>	NA	Valid	N.E.	Store RAX at address RDI or EDI.
AA	STOSB	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOSW	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOSD	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOSQ	NA	Valid	N.E.	Store RAX at address RDI or EDI.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NA	NA	NA	NA	NA

**Description**

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of the instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, EAX for doubleword operands). The explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES: (E)DI register. These must be loaded correctly before the store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, doubleword, and quadword versions of the STOS instructions. Here also ES: (E)DI is assumed to be the destination operand and AL, AX, or EAX is assumed to be the source operand. The size of the destination and source operands is selected by the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the register to the memory location, the (E)DI register is incremented or decremented according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the register is incremented; if the DF flag is 1, the register is decremented (the register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for doubleword operations).

**NOTE**

To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with STOS and STOSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

The STOS, STOSB, STOSW, STOSD, STOSQ instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

**Operation**

Non-64-bit Mode:

```
IF (Byte store)
  THEN
    DEST ← AL;
    THEN IF DF = 0
      THEN (E)DI ← (E)DI + 1;
      ELSE (E)DI ← (E)DI - 1;
    FI;
  ELSE IF (Word store)
    THEN
      DEST ← AX;
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 2;
        ELSE (E)DI ← (E)DI - 2;
      FI;
    FI;
  ELSE IF (Doubleword store)
    THEN
      DEST ← EAX;
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 4;
        ELSE (E)DI ← (E)DI - 4;
      FI;
    FI;
FI;
```

64-bit Mode:

```
IF (Byte store)
  THEN
    DEST ← AL;
    THEN IF DF = 0
      THEN (R|E)DI ← (R|E)DI + 1;
      ELSE (R|E)DI ← (R|E)DI - 1;
    FI;
  ELSE IF (Word store)
```

```

THEN
    DEST ← AX;
    THEN IF DF = 0
        THEN (R|E)DI ← (R|E)DI + 2;
        ELSE (R|E)DI ← (R|E)DI - 2;
    FI;
FI;
ELSE IF (Doubleword store)
    THEN
        DEST ← EAX;
        THEN IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 4;
            ELSE (R|E)DI ← (R|E)DI - 4;
        FI;
    FI;
ELSE IF (Quadword store using REX.W )
    THEN
        DEST ← RAX;
        THEN IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 8;
            ELSE (R|E)DI ← (R|E)DI - 8;
        FI;
    FI;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the limit of the ES segment. If the ES register contains a NULL segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the ES segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the ES segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

**STR—Store Task Register**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 00 /1	STR <i>r/m16</i>	M	Valid	Valid	Stores segment selector from TR in <i>r/m16</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

**Description**

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

**Operation**

DEST ← TR(SegmentSelector);

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD                   The STR instruction is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD                   The STR instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

- #GP(0) If the memory address is in a non-canonical form.
- #SS(U) If the stack address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

**SUB—Subtract**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2C <i>ib</i>	SUB AL, <i>imm8</i>	I	Valid	Valid	Subtract <i>imm8</i> from AL.
2D <i>iw</i>	SUB AX, <i>imm16</i>	I	Valid	Valid	Subtract <i>imm16</i> from AX.
2D <i>id</i>	SUB EAX, <i>imm32</i>	I	Valid	Valid	Subtract <i>imm32</i> from EAX.
REX.W + 2D <i>id</i>	SUB RAX, <i>imm32</i>	I	Valid	N.E.	Subtract <i>imm32</i> sign-extended to 64-bits from RAX.
80 /5 <i>ib</i>	SUB <i>r/m8, imm8</i>	MI	Valid	Valid	Subtract <i>imm8</i> from <i>r/m8</i> .
REX + 80 /5 <i>ib</i>	SUB <i>r/m8*, imm8</i>	MI	Valid	N.E.	Subtract <i>imm8</i> from <i>r/m8</i> .
81 /5 <i>iw</i>	SUB <i>r/m16, imm16</i>	MI	Valid	Valid	Subtract <i>imm16</i> from <i>r/m16</i> .
81 /5 <i>id</i>	SUB <i>r/m32, imm32</i>	MI	Valid	Valid	Subtract <i>imm32</i> from <i>r/m32</i> .
REX.W + 81 /5 <i>id</i>	SUB <i>r/m64, imm32</i>	MI	Valid	N.E.	Subtract <i>imm32</i> sign-extended to 64-bits from <i>r/m64</i> .
83 /5 <i>ib</i>	SUB <i>r/m16, imm8</i>	MI	Valid	Valid	Subtract sign-extended <i>imm8</i> from <i>r/m16</i> .
83 /5 <i>ib</i>	SUB <i>r/m32, imm8</i>	MI	Valid	Valid	Subtract sign-extended <i>imm8</i> from <i>r/m32</i> .
REX.W + 83 /5 <i>ib</i>	SUB <i>r/m64, imm8</i>	MI	Valid	N.E.	Subtract sign-extended <i>imm8</i> from <i>r/m64</i> .
28 / <i>r</i>	SUB <i>r/m8, r8</i>	MR	Valid	Valid	Subtract <i>r8</i> from <i>r/m8</i> .
REX + 28 / <i>r</i>	SUB <i>r/m8*, r8*</i>	MR	Valid	N.E.	Subtract <i>r8</i> from <i>r/m8</i> .
29 / <i>r</i>	SUB <i>r/m16, r16</i>	MR	Valid	Valid	Subtract <i>r16</i> from <i>r/m16</i> .
29 / <i>r</i>	SUB <i>r/m32, r32</i>	MR	Valid	Valid	Subtract <i>r32</i> from <i>r/m32</i> .
REX.W + 29 / <i>r</i>	SUB <i>r/m64, r64</i>	MR	Valid	N.E.	Subtract <i>r64</i> from <i>r/m64</i> .
2A / <i>r</i>	SUB <i>r8, r/m8</i>	RM	Valid	Valid	Subtract <i>r/m8</i> from <i>r8</i> .
REX + 2A / <i>r</i>	SUB <i>r8*, r/m8*</i>	RM	Valid	N.E.	Subtract <i>r/m8</i> from <i>r8</i> .
2B / <i>r</i>	SUB <i>r16, r/m16</i>	RM	Valid	Valid	Subtract <i>r/m16</i> from <i>r16</i> .
2B / <i>r</i>	SUB <i>r32, r/m32</i>	RM	Valid	Valid	Subtract <i>r/m32</i> from <i>r32</i> .
REX.W + 2B / <i>r</i>	SUB <i>r64, r/m64</i>	RM	Valid	N.E.	Subtract <i>r/m64</i> from <i>r64</i> .

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/26/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/26/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

**Description**

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

## Operation

$\text{DEST} \leftarrow (\text{DEST} - \text{SRC});$

## 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**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 bit Mode Support</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
66 0F 5C /r SUBPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Subtract packed double-precision floating-point values in <i>xmm2/m128</i> from <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 5C /r VSUBPD <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Subtract packed double-precision floating-point values in <i>xmm3/mem</i> from <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG 5C /r VSUBPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Subtract packed double-precision floating-point values in <i>ymm3/mem</i> from <i>ymm2</i> and stores result in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the two packed double-precision floating-point values in the source operand (second operand) from the two packed double-precision floating-point values in the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**Operation****SUBPD (128-bit Legacy SSE version)**

DEST[63:0]  $\leftarrow$  DEST[63:0] - SRC[63:0]  
 DEST[127:64]  $\leftarrow$  DEST[127:64] - SRC[127:64]  
 DEST[VLMAX-1:128] (Unmodified)

**VSUBPD (VEX.128 encoded version)**

DEST[63:0]  $\leftarrow$  SRC1[63:0] - SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64] - SRC2[127:64]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

**VSUBPD (VEX.256 encoded version)**

DEST[63:0]  $\leftarrow$  SRC1[63:0] - SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64] - SRC2[127:64]  
 DEST[191:128]  $\leftarrow$  SRC1[191:128] - SRC2[191:128]  
 DEST[255:192]  $\leftarrow$  SRC1[255:192] - SRC2[255:192]

**Intel C/C++ Compiler Intrinsic Equivalent**

SUBPD:       `__m128d _mm_sub_pd (__m128d a, __m128d b)`  
 VSUBPD:      `__m256d _mm256_sub_pd (__m256d a, __m256d b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2.

**SUBPS—Subtract Packed Single-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 5C /r		RM	V/V	SSE
SUBPS <i>xmm1</i> <i>xmm2/m128</i>				Subtract packed single-precision floating-point values in <i>xmm2/mem</i> from <i>xmm1</i> .
VEX.NDS.128.0F.WIG 5C /r		RVM	V/V	AVX
VSUBPS <i>xmm1,xmm2, xmm3/m128</i>				Subtract packed single-precision floating-point values in <i>xmm3/mem</i> from <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.0F.WIG 5C /r		RVM	V/V	AVX
VSUBPS <i>ymm1, ymm2, ymm3/m256</i>				Subtract packed single-precision floating-point values in <i>ymm3/mem</i> from <i>ymm2</i> and stores result in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the four packed single-precision floating-point values in the source operand (second operand) from the four packed single-precision floating-point values in the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**Operation****SUBPS (128-bit Legacy SSE version)**

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

**VSUBPS (VEX.128 encoded version)**

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

**VSUBPS (VEX.256 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] - \text{SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] - \text{SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] - \text{SRC2}[255:224].$

**Intel C/C++ Compiler Intrinsic Equivalent**

SUBPS:       `__m128 _mm_sub_ps(__m128 a, __m128 b)`  
 VSUBPS:      `__m256 _mm256_sub_ps (__m256 a, __m256 b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2.

**SUBSD—Subtract Scalar Double-Precision Floating-Point Values**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 OF 5C /r SUBSD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE2	Subtracts the low double-precision floating-point values in <i>xmm2/mem64</i> from <i>xmm1</i> .
VEX.NDS.LIG.F2.OF.WIG 5C /r VSUBSD <i>xmm1,xmm2, xmm3/m64</i>	RVM	V/V	AVX	Subtract the low double-precision floating-point value in <i>xmm3/mem</i> from <i>xmm2</i> and store the result in <i>xmm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Subtracts the low double-precision floating-point value in the source operand (second operand) from the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

**Operation****SUBSD (128-bit Legacy SSE version)**

DEST[63:0]  $\leftarrow$  DEST[63:0] - SRC[63:0]  
DEST[VLMAX-1:64] (Unmodified)

**VSUBSD (VEX.128 encoded version)**

DEST[63:0]  $\leftarrow$  SRC1[63:0] - SRC2[63:0]  
DEST[127:64]  $\leftarrow$  SRC1[127:64]  
DEST[VLMAX-1:128]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

SUBSD:  $\_\_m128d\ _{mm\_sub\_sd}(\_m128d\ a,\_m128d\ b)$

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 3.

## SUBSS—Subtract Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 5C /r SUBSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Subtract the lower single-precision floating-point values in <i>xmm2/m32</i> from <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.WIG 5C /r VSUBSS <i>xmm1,xmm2, xmm3/m32</i>	RVM	V/V	AVX	Subtract the low single-precision floating-point value in <i>xmm3/mem</i> from <i>xmm2</i> and store the result in <i>xmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Subtracts the low single-precision floating-point value in the source operand (second operand) from the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

### Operation

#### SUBSS (128-bit Legacy SSE version)

DEST[31:0]  $\leftarrow$  DEST[31:0] - SRC[31:0]  
DEST[VLMAX-1:32] (Unmodified)

#### VSUBSS (VEX.128 encoded version)

DEST[31:0]  $\leftarrow$  SRC1[31:0] - SRC2[31:0]  
DEST[127:32]  $\leftarrow$  SRC1[127:32]  
DEST[VLMAX-1:128]  $\leftarrow$  0

### Intel C/C++ Compiler Intrinsic Equivalent

SUBSS:  $\_\_m128\_\_mm\_sub\_ss(\_\_m128\ a,\_\_m128\ b)$

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Other Exceptions

See Exceptions Type 3.

## SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 F8	SWAPGS	NP	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32\_KERNEL\_GS\_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32\_KERNEL\_GS\_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32\_KERNEL\_GS\_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32\_KERNEL\_GS\_BASE MSR contains a canonical address.

### Operation

IF CS.L ≠ 1 (\* Not in 64-Bit Mode \*)

THEN

#UD; FI;

IF CPL ≠ 0

THEN #GP(0); FI;

```
tmp ← GS.base;
GS.base ← IA32_KERNEL_GS_BASE;
IA32_KERNEL_GS_BASE ← tmp;
```

### Flags Affected

None

### Protected Mode Exceptions

#UD If Mode ≠ 64-Bit.

### Real-Address Mode Exceptions

#UD If Mode ≠ 64-Bit.

### Virtual-8086 Mode Exceptions

#UD If Mode ≠ 64-Bit.

**Compatibility Mode Exceptions**

#UD                  If Mode  $\neq$  64-Bit.

**64-Bit Mode Exceptions**

#GP(0)                If CPL  $\neq$  0.  
                        If the LOCK prefix is used.

## SYSCALL—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 05	SYSCALL	NP	Valid	Invalid	Fast call to privilege level 0 system procedures.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32\_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32\_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32\_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32\_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32\_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

### Operation

IF (CS.L ≠ 1) or (IA32\_EFER.LMA ≠ 1) or (IA32\_EFER.SCE ≠ 1)

(\* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32\_EFER \*)

THEN #UD;

Fl;

RCX ← RIP; (\* Will contain address of next instruction \*)

RIP ← IA32\_LSTAR;

R11 ← RFLAGS;

RFLAGS ← RFLAGS AND NOT(IA32\_FMASK);

CS.Selector ← IA32\_STAR[47:32] AND FFFCH (\* Operating system provides CS; RPL forced to 0 \*)

(\* Set rest of CS to a fixed value \*)

CS.Base ← 0; (\* Flat segment \*)

CS.Limit ← FFFFFH; (\* With 4-KByte granularity, implies a 4-GByte limit \*)

CS.Type ← 11; (\* Execute/read code, accessed \*)

CS.S ← 1;

CS.DPL ← 0;

CS.P ← 1;

CS.L ← 1; (\* Entry is to 64-bit mode \*)

CS.D ← 0; (\* Required if CS.L = 1 \*)

CS.G ← 1; (\* 4-KByte granularity \*)

CPL ← 0;

```

SS.Selector ← IA32_STAR[47:32] + 8;      (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                          (* Flat segment *)
SS.Limit ← FFFFH;                    (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                         (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;                           (* 32-bit stack segment *)
SS.G ← 1;                           (* 4-KByte granularity *)

```

### Flags Affected

All.

### Protected Mode Exceptions

#UD                The SYSCALL instruction is not recognized in protected mode.

### Real-Address Mode Exceptions

#UD                The SYSCALL instruction is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD                The SYSCALL instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD                The SYSCALL instruction is not recognized in compatibility mode.

### 64-Bit Mode Exceptions

#UD                If IA32\_EFER.SCE = 0.

If the LOCK prefix is used.

## SYSENTER—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 34	SYSENTER	NP	Valid	Valid	Fast call to privilege level 0 system procedures.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32\_SYSENTER\_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32\_SYSENTER\_EIP** (MSR address 175H) — 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 176H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32\_SYSENTER\_EIP and IA32\_SYSENTER\_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32\_SYSENTER\_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER\_CS\_MSR MSR.
- The fast system call “stub” routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
  THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
    THEN
      SYSENTER/SYSEXIT_Not_Supported; Fl;
    ELSE
      SYSENTER/SYSEXIT_Supported; Fl;
  Fl;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

## Operation

```
IF CRO.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); Fl;

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];
  Fl;

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 ← FFFFH; (* 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*)
  Fl;
CS.G ← 1; (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← CS.Selector + 8; (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0; (* Flat segment *)
SS.Limit ← FFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3; (* Read/write data, accessed *)
```

```

SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;          (* 32-bit stack segment*)
SS.G ← 1;          (* 4-KByte granularity *)

```

### Flags Affected

VM, IF (see Operation above)

### Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	The SYSENTER instruction is not recognized in real-address mode.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 35	SYSEXIT	NP	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + OF 35	SYSEXIT	NP	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protection levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32\_SYSENTER\_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32\_SYSENTER\_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32\_SYSENTER\_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
  THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
    THEN
      SYSENTER/SYSEXIT_Not_Supported; Fl;
    ELSE
      SYSENTER/SYSEXIT_Supported; Fl;
  Fl;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

## Operation

```

IF IA32_SYSENTER_CS[15:2] = 0 OR CR0.PE = 0 OR CPL ≠ 0 THEN #GP(0); Fl;

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;
  Fl;

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;
Fl;
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 ← FFFFH;                 (* 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;                  (* 32-bit code segment*)
    CS.D ← 1;
  Fl;
CS.G ← 1;                        (* 4-KByte granularity *)
CPL ← 3;

SS.Selector ← CS.Selector + 8;      (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                      (* Flat segment *)
SS.Limit ← FFFFH;                 (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                     (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                        (* 32-bit stack segment*)
SS.G ← 1;                        (* 4-KByte granularity *)

```

## Flags Affected

None.

## Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
	If CPL ≠ 0.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

- #GP                The SYSEXIT instruction is not recognized in real-address mode.
- #UD                If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

- #GP(0)            The SYSEXIT instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #GP(0)            If IA32\_SYSENTER\_CS = 0.  
                    If CPL ≠ 0.  
                    If RCX or RDX contains a non-canonical address.
- #UD                If the LOCK prefix is used.

## SYSRET—Return From Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 07	SYSRET	NP	Valid	Invalid	Return to compatibility mode from fast system call
REX.W + OF 07	SYSRET	NP	Valid	Invalid	Return to 64-bit mode from fast system call

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.<sup>1</sup> With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32\_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, “Interrupt Stack Table,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).
- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - Confirming that the value of RCX is canonical before executing SYSRET.
  - Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - Using the IST mechanism for gate 13 (#GP) in the IDT.

### Operation

IF (CS.L ≠ 1) or (IA32\_EFER.LMA ≠ 1) or (IA32\_EFER.SCE ≠ 1)

(\* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32\_EFER \*)

THEN #UD; FI;

IF (CPL ≠ 0) OR (RCX is not canonical) THEN #GP(0); FI;

1. Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.

```

IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP ← ECX;
    Fi;
    RFLAGS ← (R11 & 3C7FD7H) | 2;           (* Clear RF, VM, reserved bits; set bit 2 *)

IF (operand size is 64-bit)
    THEN CS.Selector ← IA32_STAR[63:48]+16;
    ELSE CS.Selector ← IA32_STAR[63:48];
Fi;
CS.Selector ← CS.Selector OR 3;          (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                            (* Flat segment *)
CS.Limit ← FFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                           (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;

IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        CS.L ← 1;                         (* 64-bit code segment *)
        CS.D ← 0;                         (* Required if CS.L = 1 *)
    ELSE (* Return to Compatibility Mode *)
        CS.L ← 0;                         (* Compatibility mode *)
        CS.D ← 1;                         (* 32-bit code segment *)
    Fi;
CS.G ← 1;                             (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← (IA32_STAR[63:48]+8) OR 3; (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                            (* Flat segment *)
SS.Limit ← FFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                           (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                            (* 32-bit stack segment*)
SS.G ← 1;                            (* 4-KByte granularity *)

```

## Flags Affected

All.

## Protected Mode Exceptions

#UD                  The SYSRET instruction is not recognized in protected mode.

## Real-Address Mode Exceptions

#UD                  The SYSRET instruction is not recognized in real-address mode.

## Virtual-8086 Mode Exceptions

#UD                  The SYSRET instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD                    The SYSRET instruction is not recognized in compatibility mode.

### 64-Bit Mode Exceptions

#UD                    If IA32\_EFER.SCE = 0.

                        If the LOCK prefix is used.

#GP(0)                If CPL ≠ 0.

                        If RCX contains a non-canonical address.

## TEST—Logical Compare

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A8 <i>ib</i>	TEST AL, <i>imm8</i>	I	Valid	Valid	AND <i>imm8</i> with AL; set SF, ZF, PF according to result.
A9 <i>iw</i>	TEST AX, <i>imm16</i>	I	Valid	Valid	AND <i>imm16</i> with AX; set SF, ZF, PF according to result.
A9 <i>id</i>	TEST EAX, <i>imm32</i>	I	Valid	Valid	AND <i>imm32</i> with EAX; set SF, ZF, PF according to result.
REX.W + A9 <i>id</i>	TEST RAX, <i>imm32</i>	I	Valid	N.E.	AND <i>imm32</i> sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 <i>ib</i>	TEST <i>r/m8, imm8</i>	MI	Valid	Valid	AND <i>imm8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
REX + F6 /0 <i>ib</i>	TEST <i>r/m8*, imm8</i>	MI	Valid	N.E.	AND <i>imm8</i> with <i>r/m8*</i> ; set SF, ZF, PF according to result.
F7 /0 <i>iw</i>	TEST <i>r/m16, imm16</i>	MI	Valid	Valid	AND <i>imm16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result.
F7 /0 <i>id</i>	TEST <i>r/m32, imm32</i>	MI	Valid	Valid	AND <i>imm32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result.
REX.W + F7 /0 <i>id</i>	TEST <i>r/m64, imm32</i>	MI	Valid	N.E.	AND <i>imm32</i> sign-extended to 64-bits with <i>r/m64</i> ; set SF, ZF, PF according to result.
84 / <i>r</i>	TEST <i>r/m8, r8</i>	MR	Valid	Valid	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
REX + 84 / <i>r</i>	TEST <i>r/m8*, r8*</i>	MR	Valid	N.E.	AND <i>r8</i> with <i>r/m8*</i> ; set SF, ZF, PF according to result.
85 / <i>r</i>	TEST <i>r/m16, r16</i>	MR	Valid	Valid	AND <i>r16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result.
85 / <i>r</i>	TEST <i>r/m32, r32</i>	MR	Valid	Valid	AND <i>r32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result.
REX.W + 85 / <i>r</i>	TEST <i>r/m64, r64</i>	MR	Valid	N.E.	AND <i>r64</i> with <i>r/m64</i> ; set SF, ZF, PF according to result.

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r)	imm8/16/32	NA	NA
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA

### Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);
```

```
IF TEMP = 0
  THEN ZF ← 1;
  ELSE ZF ← 0;
```

Fl:

```
PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(* AF is undefined *)
```

## Flags Affected

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

## Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## TZCNT – Count the Number of Trailing Zero Bits

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 OF BC /r TZCNT r16, r/m16	RM	V/V	BMI1	Count the number of trailing zero bits in r/m16, return result in r16.
F3 OF BC /r TZCNT r32, r/m32	RM	V/V	BMI1	Count the number of trailing zero bits in r/m32, return result in r32.
REX.W + F3 OF BC /r TZCNT r64, r/m64	RM	V/N.E.	BMI1	Count the number of trailing zero bits in r/m64, return result in r64.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between TZCNT and BSF instruction is that TZCNT provides operand size as output when source operand is zero while in the case of BSF instruction, if source operand is zero, the content of destination operand are undefined. On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

### Operation

```
temp ← 0
DEST ← 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp ] = 0) )
```

```
    temp ← temp +1
    DEST ← DEST+ 1
OD
```

```
IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI
```

```
IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI
```

### Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**

```
TZCNT:     unsigned __int32 _tzcnt_u32(unsigned __int32 src);
TZCNT:     unsigned __int64 _tzcnt_u64(unsigned __int64 src);
```

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.

**Virtual 8086 Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 2E /r UCOMISD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE2	Compares (unordered) the low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/m64</i> and set the EFLAGS accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Compare low double precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

**Description**

Performs an unordered compare of the double-precision floating-point values in the low quadwords of source operand 1 (first operand) and source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

```

RESULT ← UnorderedCompare(SRC1[63:0] < > SRC2[63:0]) {
(* Set EFLAGS *)
CASE (RESULT) OF
  UNORDERED:      ZF, PF, CF ← 111;
  GREATER_THAN:   ZF, PF, CF ← 000;
  LESS_THAN:      ZF, PF, CF ← 001;
  EQUAL:          ZF, PF, CF ← 100;
ESAC;
OF, AF, SF ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

int _mm_ucomieq_sd(__m128d a, __m128d b)
int _mm_ucomilt_sd(__m128d a, __m128d b)
int _mm_ucomile_sd(__m128d a, __m128d b)
int _mm_ucomigt_sd(__m128d a, __m128d b)
int _mm_ucomige_sd(__m128d a, __m128d b)
int _mm_ucomineq_sd(__m128d a, __m128d b)

```

### SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

### Other Exceptions

See Exceptions Type 3; additionally

#UD                    If VEX.vvvv != 1111B.

**UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 2E /r UCOMISS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Compare lower single-precision floating-point value in <i>xmm1</i> register with lower single-precision floating-point value in <i>xmm2/mem</i> and set the status flags accordingly.
VEX.LIG.OF.WIG 2E /r VUCOMISS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	AVX	Compare low single precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

**Description**

Performs an unordered compare of the single-precision floating-point values in the low doublewords of the source operand 1 (first operand) and the source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISS instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

```
RESULT ← UnorderedCompare(SRC1[31:0] <> SRC2[31:0]) {
(* Set EFLAGS *)
CASE (RESULT) OF
  UNORDERED:      ZF,PF,CF ← 111;
  GREATER_THAN:   ZF,PF,CF ← 000;
  LESS_THAN:      ZF,PF,CF ← 001;
  EQUAL:          ZF,PF,CF ← 100;
ESAC;
OF,AF,SF ← 0;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
int _mm_ocomieq_ss(__m128 a, __m128 b)
int _mm_ocomilt_ss(__m128 a, __m128 b)
int _mm_ocomile_ss(__m128 a, __m128 b)
int _mm_ocomigt_ss(__m128 a, __m128 b)
int _mm_ocomige_ss(__m128 a, __m128 b)
```

int \_mm\_ucomineq\_ss(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

### Other Exceptions

See Exceptions Type 3; additionally

#UD                  If VEX.vvvv != 1111B.

## UD2—Undefined Instruction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 0B	UD2	NP	Valid	Valid	Raise invalid opcode exception.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcode for this instruction is reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD2 instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD2 instruction (and not the following instruction).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

#UD (\* Generates invalid opcode exception \*);

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD                    Raises an invalid opcode exception in all operating modes.

**UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values**

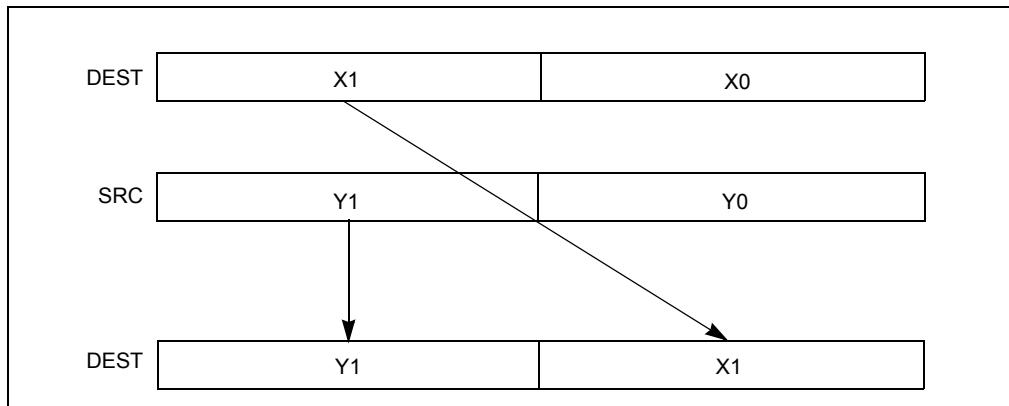
Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of <i>xmm1</i> and <i>xmm2/m128</i> .
VEX.NDS.128.66.0F.WIG 15 /r VUNPCKHPD <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of <i>xmm2</i> and <i>xmm3/m128</i> .
VEX.NDS.256.66.0F.WIG 15 /r VUNPCKHPD <i>ymm1,ymm2, ymm3/m256</i>	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of <i>ymm2</i> and <i>ymm3/m256</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs an interleaved unpack of the high double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-23.



**Figure 4-23. UNPCKHPD Instruction High Unpack and Interleave Operation**

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

## Operation

### **UNPCKHPD (128-bit Legacy SSE version)**

DEST[63:0]  $\leftarrow$  SRC1[127:64]  
 DEST[127:64]  $\leftarrow$  SRC2[127:64]  
 DEST[VLMAX-1:128] (Unmodified)

### **VUNPCKHPD (VEX.128 encoded version)**

DEST[63:0]  $\leftarrow$  SRC1[127:64]  
 DEST[127:64]  $\leftarrow$  SRC2[127:64]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VUNPCKHPD (VEX.256 encoded version)**

DEST[63:0]  $\leftarrow$  SRC1[127:64]  
 DEST[127:64]  $\leftarrow$  SRC2[127:64]  
 DEST[191:128]  $\leftarrow$  SRC1[255:192]  
 DEST[255:192]  $\leftarrow$  SRC2[255:192]

## Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD: `__m128d _mm_unpackhi_pd(__m128d a, __m128d b)`  
 UNPCKHPD: `__m256d _mm256_unpackhi_pd(__m256d a, __m256d b)`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4.

**UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values**

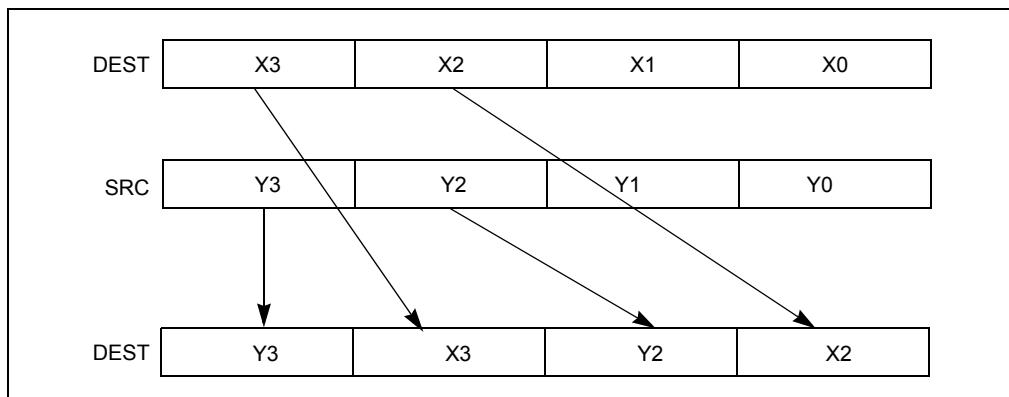
Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 15 /r UNPCKHPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of <i>xmm1</i> and <i>xmm2/mem</i> into <i>xmm1</i> .
VEX.NDS.128.0F.WIG 15 /r VUNPCKHPS <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of <i>xmm2</i> and <i>xmm3/m128</i> .
VEX.NDS.256.0F.WIG 15 /r VUNPCKHPS <i>ymm1,ymm2,ymm3/m256</i>	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of <i>ymm2</i> and <i>ymm3/m256</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs an interleaved unpack of the high-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-24. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 4-24. UNPCKHPS Instruction High Unpack and Interleave Operation**

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

## Operation

### **UNPCKHPS (128-bit Legacy SSE version)**

DEST[31:0]  $\leftarrow$  SRC1[95:64]  
 DEST[63:32]  $\leftarrow$  SRC2[95:64]  
 DEST[95:64]  $\leftarrow$  SRC1[127:96]  
 DEST[127:96]  $\leftarrow$  SRC2[127:96]  
 DEST[VLMAX-1:128] (Unmodified)

### **VUNPCKHPS (VEX.128 encoded version)**

DEST[31:0]  $\leftarrow$  SRC1[95:64]  
 DEST[63:32]  $\leftarrow$  SRC2[95:64]  
 DEST[95:64]  $\leftarrow$  SRC1[127:96]  
 DEST[127:96]  $\leftarrow$  SRC2[127:96]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VUNPCKHPS (VEX.256 encoded version)**

DEST[31:0]  $\leftarrow$  SRC1[95:64]  
 DEST[63:32]  $\leftarrow$  SRC2[95:64]  
 DEST[95:64]  $\leftarrow$  SRC1[127:96]  
 DEST[127:96]  $\leftarrow$  SRC2[127:96]  
 DEST[159:128]  $\leftarrow$  SRC1[223:192]  
 DEST[191:160]  $\leftarrow$  SRC2[223:192]  
 DEST[223:192]  $\leftarrow$  SRC1[255:224]  
 DEST[255:224]  $\leftarrow$  SRC2[255:224]

## Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPS: `__m128_mm_unpackhi_ps(__m128 a, __m128 b)`  
 UNPCKHPS: `__m256_mm256_unpackhi_ps (__m256 a, __m256 b);`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4.

**UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values**

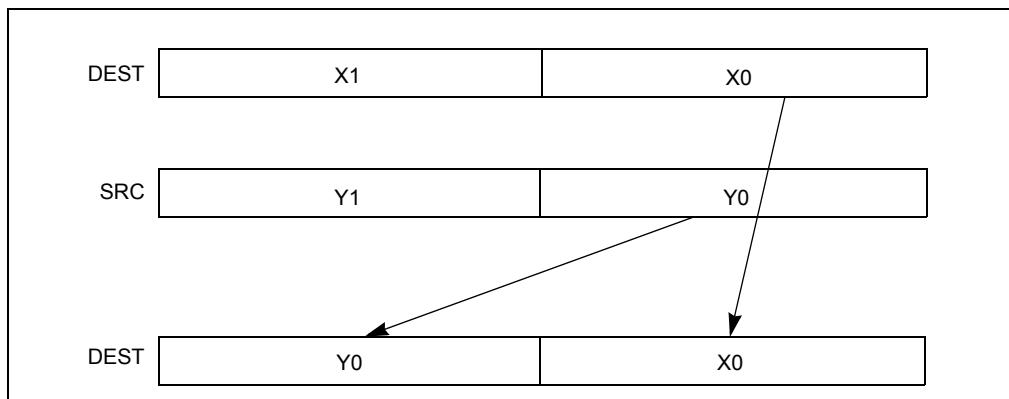
Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from low quadwords of <i>xmm1</i> and <i>xmm2/m128</i> .
VEX.NDS.128.66.0F.WIG 14 /r VUNPCKLPD <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values low high quadwords of <i>xmm2</i> and <i>xmm3/m128</i> .
VEX.NDS.256.66.0F.WIG 14 /r VUNPCKLPD <i>ymm1,ymm2, ymm3/m256</i>	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values low high quadwords of <i>ymm2</i> and <i>ymm3/m256</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs an interleaved unpack of the low double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-25. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 4-25. UNPCKLPD Instruction Low Unpack and Interleave Operation**

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

## Operation

### **UNPCKLPD (128-bit Legacy SSE version)**

DEST[63:0]  $\leftarrow$  SRC1[63:0]  
 DEST[127:64]  $\leftarrow$  SRC2[63:0]  
 DEST[VLMAX-1:128] (Unmodified)

### **VUNPCKLPD (VEX.128 encoded version)**

DEST[63:0]  $\leftarrow$  SRC1[63:0]  
 DEST[127:64]  $\leftarrow$  SRC2[63:0]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

### **VUNPCKLPD (VEX.256 encoded version)**

DEST[63:0]  $\leftarrow$  SRC1[63:0]  
 DEST[127:64]  $\leftarrow$  SRC2[63:0]  
 DEST[191:128]  $\leftarrow$  SRC1[191:128]  
 DEST[255:192]  $\leftarrow$  SRC2[191:128]

## Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD: `__m128d _mm_unpacklo_pd(__m128d a, __m128d b)`  
 UNPCKLPD: `__m256d _mm256_unpacklo_pd(__m256d a, __m256d b)`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4.

**UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values**

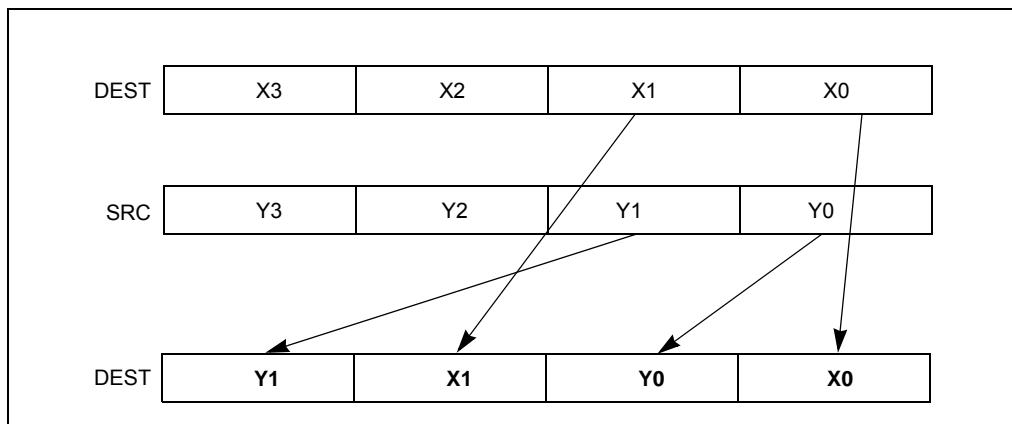
Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 14 /r UNPCKLPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of <i>xmm1</i> and <i>xmm2/mem</i> into <i>xmm1</i> .
VEX.NDS.128.0F.WIG 14 /r VUNPCKLPS <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of <i>xmm2</i> and <i>xmm3/m128</i> .
VEX.NDS.256.0F.WIG 14 /r VUNPCKLPS <i>ymm1,ymm2,ymm3/m256</i>	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of <i>ymm2</i> and <i>ymm3/m256</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs an interleaved unpack of the low-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-26. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 4-26. UNPCKLPS Instruction Low Unpack and Interleave Operation**

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

**VEX.128 encoded version:** The first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

## Operation

### UNPCKLPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[VLMAX-1:128] (Unmodified)

### VUNPCKLPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]

DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]

DEST[VLMAX-1:128] ← 0

### 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]

## Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPS: `__m128_mm_unpacklo_ps(__m128 a, __m128 b)`

UNPCKLPS: `__m256_mm256_unpacklo_ps (__m256 a, __m256 b);`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4.

**VBROADCAST—Broadcast Floating-Point Data**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32-bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
VEX.128.66.0F38.W0 18 /r VBROADCASTSS <i>xmm1, m32</i>	RM	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 18 /r VBROADCASTSS <i>ymm1, m32</i>	RM	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in <i>ymm1</i> .
VEX.256.66.0F38.W0 19 /r VBROADCASTSD <i>ymm1, m64</i>	RM	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in <i>ymm1</i> .
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 <i>ymm1, m128</i>	RM	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in <i>ymm1</i> .
VEX.128.66.0F38.W0 18/r VBROADCASTSS <i>xmm1, xmm2</i>	RM	V/V	AVX2	Broadcast the low single-precision floating-point element in the source operand to four locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 18 /r VBROADCASTSS <i>ymm1, xmm2</i>	RM	V/V	AVX2	Broadcast low single-precision floating-point element in the source operand to eight locations in <i>ymm1</i> .
VEX.256.66.0F38.W0 19 /r VBROADCASTSD <i>ymm1, xmm2</i>	RM	V/V	AVX2	Broadcast low double-precision floating-point element in the source operand to four locations in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Load floating point values from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

VBROADCASTSD and VBROADCASTF128 are only supported as 256-bit wide versions. VBROADCASTSS is supported in both 128-bit and 256-bit wide versions.

Memory and register source operand syntax support of 256-bit instructions depend on the processor's enumeration of the following conditions with respect to CPUID.1:ECX.AVX[bit 28] and CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]:

- If CPUID.1:ECX.AVX = 1 and CPUID.(EAX=07H, ECX=0H):EBX.AVX2 = 0: the destination operand is a YMM register. The source operand support can be either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD.
- If CPUID.1:ECX.AVX = 1 and CPUID.(EAX=07H, ECX=0H):EBX.AVX2 = 1: the destination operand is a YMM register. The source operand support can be a register or memory location.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. An attempt to execute VBROADCASTSD or VBROADCASTF128 encoded with VEX.L= 0 will cause an #UD exception. Attempts to execute any VBROADCAST\* instruction with VEX.W = 1 will cause #UD.

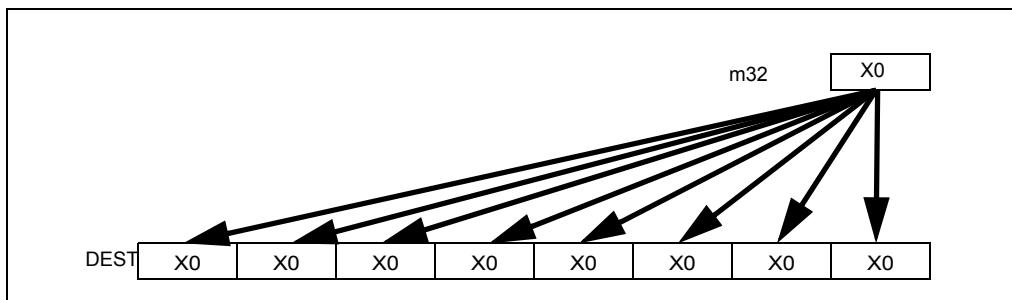


Figure 4-27. VBROADCASTSS Operation (VEX.256 encoded version)

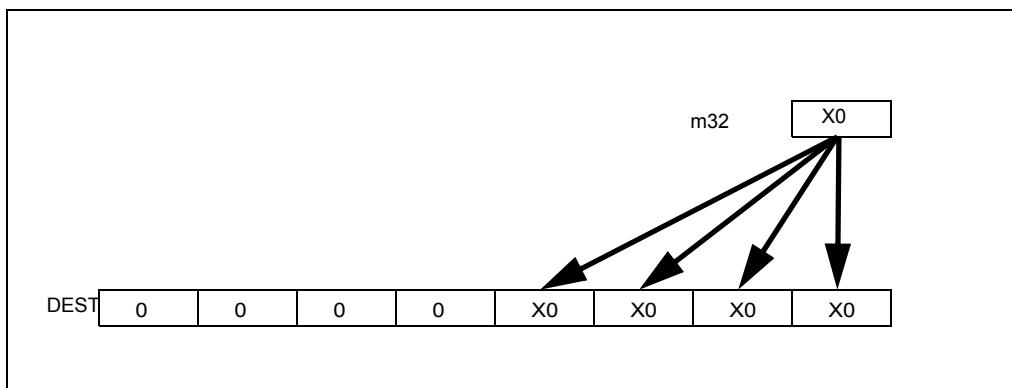


Figure 4-28. VBROADCASTSS Operation (128-bit version)

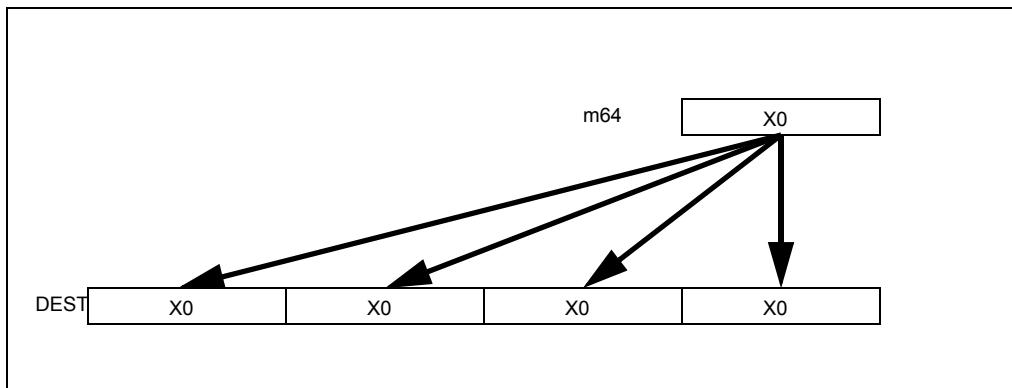


Figure 4-29. VBROADCASTSD Operation

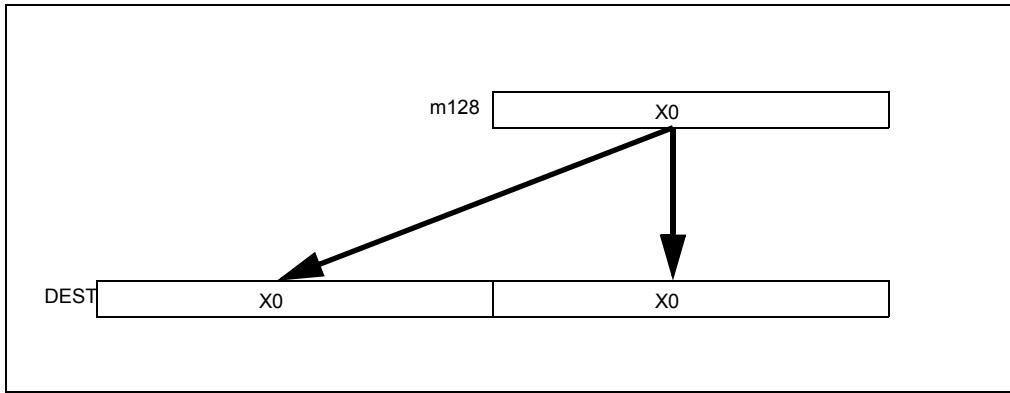


Figure 4-30. VBROADCASTF128 Operation

## Operation

### VBROADCASTSS (128 bit version)

```
temp <- SRC[31:0]
DEST[31:0] <- temp
DEST[63:32] <- temp
DEST[95:64] <- temp
DEST[127:96] <- temp
DEST[VLMAX-1:128] <- 0
```

### VBROADCASTSS (VEX.256 encoded version)

```
temp <- SRC[31:0]
DEST[31:0] <- temp
DEST[63:32] <- temp
DEST[95:64] <- temp
DEST[127:96] <- temp
DEST[159:128] <- temp
DEST[191:160] <- temp
DEST[223:192] <- temp
DEST[255:224] <- temp
```

### VBROADCASTSD (VEX.256 encoded version)

```
temp <- SRC[63:0]
DEST[63:0] <- temp
DEST[127:64] <- temp
DEST[191:128] <- temp
DEST[255:192] <- temp
```

### VBROADCASTF128

```
temp <- SRC[127:0]
DEST[127:0] <- temp
DEST[VLMAX-1:128] <- temp
```

## Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTSS:	<code>__m128 _mm_broadcast_ss(float *a);</code>
VBROADCASTSS:	<code>__m256 _mm256_broadcast_ss(float *a);</code>
VBROADCASTSD:	<code>__m256d _mm256_broadcast_sd(double *a);</code>

VBROADCASTF128: `__m256 _mm256_broadcast_ps(__m128 * a);`  
VBROADCASTF128: `__m256d _mm256_broadcast_pd(__m128d * a);`

### Flags Affected

None.

### Other Exceptions

See Exceptions Type 6; additionally

#UD                    If VEX.L = 0 for VBROADCASTSD,  
                      If VEX.L = 0 for VBROADCASTF128,  
                      If VEX.W = 1.

**VCVTPH2PS—Convert 16-bit FP Values to Single-Precision FP Values**

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 13 /r VCVTPH2PS <i>ymm1, xmm2/m128</i>	RM	V/V	F16C	Convert eight packed half precision (16-bit) floating-point values in <i>xmm2/m128</i> to packed single-precision floating-point value in <i>ymm1</i> .
VEX.128.66.0F38.W0 13 /r VCVTPH2PS <i>xmm1, xmm2/m64</i>	RM	V/V	F16C	Convert four packed half precision (16-bit) floating-point values in <i>xmm2/m64</i> to packed single-precision floating-point value in <i>xmm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Converts four/eight packed half precision (16-bits) floating-point values in the low-order 64/128 bits of an XMM/YMM register or 64/128-bit memory location to four/eight packed single-precision floating-point values and writes the converted values into the destination XMM/YMM register.

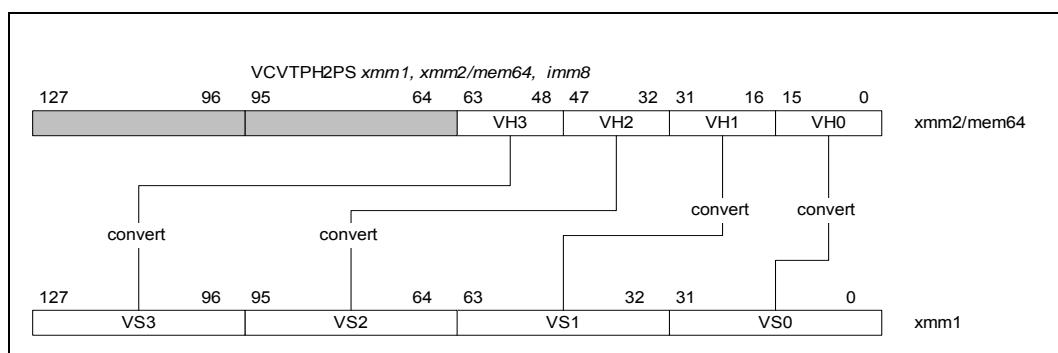
If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

128-bit version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (VLMAX-1:128) of the corresponding destination YMM register are zeroed.

256-bit version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) FP values.

Note: VEX.vvvv is reserved (must be 1111b).



**Figure 4-31. VCVTPH2PS (128-bit Version)**

**Operation**

```
vCvt_h2s(SRC1[15:0])
{
RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}
```

**VCVTPH2PS (VEX.256 encoded version)**

```

DEST[31:0] ← vCvt_h2s(SRC1[15:0]);
DEST[63:32] ← vCvt_h2s(SRC1[31:16]);
DEST[95:64] ← vCvt_h2s(SRC1[47:32]);
DEST[127:96] ← vCvt_h2s(SRC1[63:48]);
DEST[159:128] ← vCvt_h2s(SRC1[79:64]);
DEST[191:160] ← vCvt_h2s(SRC1[95:80]);
DEST[223:192] ← vCvt_h2s(SRC1[111:96]);
DEST[255:224] ← vCvt_h2s(SRC1[127:112]);

```

**VCVTPH2PS (VEX.128 encoded version)**

```

DEST[31:0] ← vCvt_h2s(SRC1[15:0]);
DEST[63:32] ← vCvt_h2s(SRC1[31:16]);
DEST[95:64] ← vCvt_h2s(SRC1[47:32]);
DEST[127:96] ← vCvt_h2s(SRC1[63:48]);
DEST[VLMAX-1:128] ← 0

```

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

```

__m128 _mm_cvtph_ps ( __m128i m1);
__m256 _mm256_cvtph_ps ( __m128i m1)

```

**SIMD Floating-Point Exceptions**

Invalid

**Other Exceptions**

Exceptions Type 11 (do not report #AC); additionally  
#UD                    If VEX.W=1.

**VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value**

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH <i>xmm1/m128, ymm2, imm8</i>	MR	V/V	F16C	Convert eight packed single-precision floating-point value in <i>ymm2</i> to packed half-precision (16-bit) floating-point value in <i>xmm1/mem</i> . <i>Imm8</i> provides rounding controls.
VEX.128.66.0F3A.W0.1D /r ib VCVTPS2PH <i>xmm1/m64, xmm2, imm8</i>	MR	V/V	F16C	Convert four packed single-precision floating-point value in <i>xmm2</i> to packed half-precision (16-bit) floating-point value in <i>xmm1/mem</i> . <i>Imm8</i> provides rounding controls.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

**Description**

Convert four or eight packed single-precision floating values in first source operand to four or eight packed half-precision (16-bit) floating-point values. The rounding mode is specified using the immediate field (*imm8*).

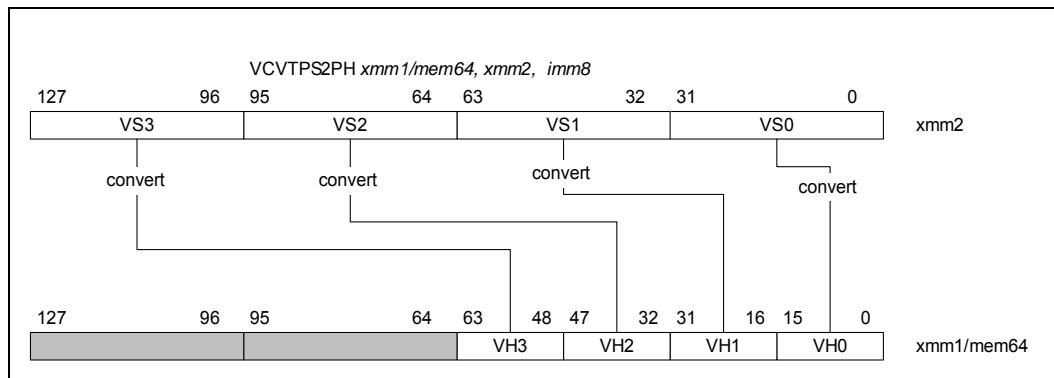
Underflow results (i.e. tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to input format with MXCSR.DAZ not set, DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

128-bit version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. The upper-bits vector register zeroing behavior of VEX prefix encoding still applies if the destination operand is a *xmm* register. So the upper bits (255:64) of corresponding YMM register are zeroed.

256-bit version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. The upper-bits vector register zeroing behavior of VEX prefix encoding still applies if the destination operand is a *xmm* register. So the upper bits (255:128) of the corresponding YMM register are zeroed.

Note: VEX.vvvv is reserved (must be 1111b).

The diagram below illustrates how data is converted from four packed single precision (in 128 bits) to four half precision (in 64 bits) FP values.



**Figure 4-32. VCVTPS2PH (128-bit Version)**

The immediate byte defines several bit fields that controls rounding operation. The effect and encoding of RC field are listed in Table 4-16.

**Table 4-16. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions**

Bits	Field Name/value	Description	Comment
Imm[1:0]	RC=00B	Round to nearest even	If Imm[2] = 0
	RC=01B	Round down	
	RC=10B	Round up	
	RC=11B	Truncate	
Imm[2]	MS1=0	Use imm[1:0] for rounding	Ignore MXCSR.RC
	MS1=1	Use MXCSR.RC for rounding	
Imm[7:3]	Ignored	Ignored by processor	

### Operation

```
vCvt_s2h(SRC1[31:0])
{
IF Imm[2] = 0
THEN // using Imm[1:0] for rounding control, see Table 4-16
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
ELSE // using MXCSR.RC for rounding control
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
FI;
}
```

### VCVTPS2PH (VEX.256 encoded version)

```
DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[79:64] ← vCvt_s2h(SRC1[159:128]);
DEST[95:80] ← vCvt_s2h(SRC1[191:160]);
DEST[111:96] ← vCvt_s2h(SRC1[223:192]);
DEST[127:112] ← vCvt_s2h(SRC1[255:224]);
DEST[255:128] ← 0; // if DEST is a register
```

### VCVTPS2PH (VEX.128 encoded version)

```
DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[VLMAX-1:64] ← 0; // if DEST is a register
```

### Flags Affected

None

**Intel C/C++ Compiler Intrinsic Equivalent**

```
_m128i_mm_cvtps_ph(_m128 m1, const int imm);  
_m128i_mm256_cvtps_ph(_m256 m1, const int imm);
```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);

**Other Exceptions**

Exceptions Type 11 (do not report #AC); additionally

#UD If VEX.W=1.

## VERR/VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 00 /4	VERR <i>r/m16</i>	M	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be read.
OF 00 /5	VERW <i>r/m16</i>	M	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be written.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment's DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

### Operation

```
IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
    THEN ZF ← 0; Fl;
```

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;

        Fl;

Fl;

## Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

## Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real-address mode. If the LOCK prefix is used.
-----	--

## Virtual-8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode. If the LOCK prefix is used.
-----	--

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## VEXTRACTF128 – Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEXTRACTF128 <i>xmm1/m128, ymm2, imm8</i>	MR	V/V	AVX	Extract 128 bits of packed floating-point values from <i>ymm2</i> and store results in <i>xmm1/mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Extracts 128-bits of packed floating-point values from the source operand (second operand) at an 128-bit offset from *imm8[0]* into the destination operand (first operand). The destination may be either an XMM register or an 128-bit memory location.

VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### VEXTRACTF128 (memory destination form)

CASE (*imm8[0]*) OF

- 0: DEST[127:0] ← SRC1[127:0]
- 1: DEST[127:0] ← SRC1[255:128]

ESAC.

#### VEXTRACTF128 (register destination form)

CASE (*imm8[0]*) OF

- 0: DEST[127:0] ← SRC1[127:0]
- 1: DEST[127:0] ← SRC1[255:128]

ESAC.

DEST[VLMAX-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

```
VEXTRACTF128:    __m128 _mm256_extractf128_ps (__m256 a, int offset);
VEXTRACTF128:    __m128d _mm256_extractf128_pd (__m256d a, int offset);
VEXTRACTF128:    __m128i_mm256_extractf128_si256(__m256i a, int offset);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 6; additionally

- #UD If VEX.L= 0
- If VEX.W=1.

**VEXTRACTI128 – Extract packed Integer Values**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 <i>xmm1/m128, ymm2, imm8</i>	RMI	V/V	AVX2	Extract 128 bits of integer data from <i>ymm2</i> and store results in <i>xmm1/mem</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

**Description**

Extracts 128-bits of packed integer values from the source operand (second operand) at a 128-bit offset from *imm8[0]* into the destination operand (first operand). The destination may be either an XMM register or a 128-bit memory location.

VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits of the immediate are ignored.

An attempt to execute VEXTRACTI128 encoded with VEX.L= 0 will cause an #UD exception.

**Operation****VEXTRACTI128 (memory destination form)**

CASE (*imm8[0]*) OF

  0: DEST[127:0] ← SRC1[127:0]

  1: DEST[127:0] ← SRC1[255:128]

ESAC.

**VEXTRACTI128 (register destination form)**

CASE (*imm8[0]*) OF

  0: DEST[127:0] ← SRC1[127:0]

  1: DEST[127:0] ← SRC1[255:128]

ESAC.

DEST[VLMAX-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXTRACTI128: \_\_m128i \_mm256\_extracti128\_si256(\_\_m256i a, int offset);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 6; additionally

#UD                    IF VEX.L = 0,  
                      If VEX.W = 1.

## VFMADD132PD/VFMADD213PD/VFMADD231PD – Fused Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 98 /r VFMADD132PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 A8 /r VFMADD213PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 B8 /r VFMADD231PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , add to <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 98 /r VFMADD132PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , add to <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 A8 /r VFMADD213PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , add to <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 B8 /r VFMADD231PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , add to <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## Operation

In the operations below, "+" , "-" , and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFMADD132PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI

```

### **VFMADD213PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI

```

### **VFMADD231PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0

```

Fl

### Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PD: `__m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);`  
VFMADD213PD: `__m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);`  
VFMADD231PD: `__m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);`  
VFMADD132PD: `__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);`  
VFMADD213PD: `__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);`  
VFMADD231PD: `__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);`

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

See Exceptions Type 2

## VFMADD132PS/VFMADD213PS/VFMADD231PS – Fused Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 98 /r VFMADD132PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 A8 /r VFMADD213PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 B8 /r VFMADD231PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , add to <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 98 /r VFMADD132PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , add to <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 A8 /r VFMADD213PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , add to <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 B8 /r VFMADD231PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , add to <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1”.

## Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFMADD132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFMADD213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFMADD231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADD132PS: `__m128_mm_fmadd_ps (__m128 a, __m128 b, __m128 c);`  
VFMADD213PS: `__m128_mm_fmadd_ps (__m128 a, __m128 b, __m128 c);`  
VFMADD231PS: `__m128_mm_fmadd_ps (__m128 a, __m128 b, __m128 c);`  
VFMADD132PS: `__m256_mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);`  
VFMADD213PS: `__m256_mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);`  
VFMADD231PS: `__m256_mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2

## VFMADD132SD/VFMADD213SD/VFMADD231SD — Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 99 /r VFMADD132SD $xmm_0, xmm_1, xmm_2/m64$	A	V/V	FMA	Multiply scalar double-precision floating-point value from $xmm_0$ and $xmm_2/mem$ , add to $xmm_1$ and put result in $xmm_0$ .
VEX.DDS.LIG.128.66.0F38.W1 A9 /r VFMADD213SD $xmm_0, xmm_1, xmm_2/m64$	A	V/V	FMA	Multiply scalar double-precision floating-point value from $xmm_0$ and $xmm_1$ , add to $xmm_2/mem$ and put result in $xmm_0$ .
VEX.DDS.LIG.128.66.0F38.W1 B9 /r VFMADD231SD $xmm_0, xmm_1, xmm_2/m64$	A	V/V	FMA	Multiply scalar double-precision floating-point value from $xmm_1$ and $xmm_2/mem$ , add to $xmm_0$ and put result in $xmm_0$ .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-add computation on the low packed double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm\_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

**VFMADD132SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[63:0]*\text{SRC3}[63:0] + \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VFMADD213SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[63:0]*\text{DEST}[63:0] + \text{SRC3}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VFMADD231SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[63:0]*\text{SRC3}[63:0] + \text{DEST}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADD132SD: `__m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);`  
VFMADD213SD: `__m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);`  
VFMADD231SD: `__m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 3

## VFMADD132SS/VFMADD213SS/VFMADD231SS – Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 99 /r VFMADD132SS $xmm_0, xmm_1, xmm_2/m32$	A	V/V	FMA	Multiply scalar single-precision floating-point value from $xmm_0$ and $xmm_2/mem$ , add to $xmm_1$ and put result in $xmm_0$ .
VEX.DDS.LIG.128.66.0F38.W0 A9 /r VFMADD213SS $xmm_0, xmm_1, xmm_2/m32$	A	V/V	FMA	Multiply scalar single-precision floating-point value from $xmm_0$ and $xmm_1$ , add to $xmm_2/mem$ and put result in $xmm_0$ .
VEX.DDS.LIG.128.66.0F38.W0 B9 /r VFMADD231SS $xmm_0, xmm_1, xmm_2/m32$	A	V/V	FMA	Multiply scalar single-precision floating-point value from $xmm_1$ and $xmm_2/mem$ , add to $xmm_0$ and put result in $xmm_0$ .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm\_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

**VFMADD132SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[31:0]*\text{SRC3}[31:0] + \text{SRC2}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VFMADD213SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[31:0]*\text{DEST}[31:0] + \text{SRC3}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VFMADD231SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[31:0]*\text{SRC3}[63:0] + \text{DEST}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADD132SS: `__m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);`  
VFMADD213SS: `__m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);`  
VFMADD231SS: `__m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 3

## VFMADDSSUB132PD/VFMADDSSUB213PD/VFMADDSSUB231PD – Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSSUB132PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , add/subtract elements in <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSSUB213PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , add/subtract elements in <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSSUB231PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , add/subtract elements in <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSSUB132PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , add/subtract elements in <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSSUB213PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , add/subtract elements in <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSSUB231PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , add/subtract elements in <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFMADDSSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### VFMADDSSUB132PD DEST, SRC2, SRC3

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192])
FI
```

### VFMADDSSUB213PD DEST, SRC2, SRC3

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192])
FI
```

### VFMADDSSUB231PD DEST, SRC2, SRC3

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192])
FI
```

## Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDSSUB132PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSSUB213PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSSUB231PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSSUB132PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);
VFMADDSSUB213PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);
```

VFMADDSUB231PD: \_\_m256d \_mm256\_fmaddsub\_pd (\_\_m256d a, \_\_m256d b, \_\_m256d c);

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

See Exceptions Type 2

## VFMADDSSUB132PS/VFMADDSSUB213PS/VFMADDSSUB231PS – Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSSUB132PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , add/subtract elements in <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSSUB213PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , add/subtract elements in <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSSUB231PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , add/subtract elements in <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSSUB132PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , add/subtract elements in <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSSUB213PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , add/subtract elements in <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSSUB231PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , add/subtract elements in <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFMADDSSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## Operation

In the operations below, "+" , "-" , and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### VFMADDSSUB132PS DEST, SRC2, SRC3

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### VFMADDSSUB213PS DEST, SRC2, SRC3

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### VFMADDSSUB231PS DEST, SRC2, SRC3

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADDSSUB132PS: `_m128_mm_fmaddsub_ps (_m128 a, _m128 b, _m128 c);`  
VFMADDSSUB213PS: `_m128_mm_fmaddsub_ps (_m128 a, _m128 b, _m128 c);`  
VFMADDSSUB231PS: `_m128_mm_fmaddsub_ps (_m128 a, _m128 b, _m128 c);`  
VFMADDSSUB132PS: `_m256_mm256_fmaddsub_ps (_m256 a, _m256 b, _m256 c);`  
VFMADDSSUB213PS: `_m256_mm256_fmaddsub_ps (_m256 a, _m256 b, _m256 c);`  
VFMADDSSUB231PS: `_m256_mm256_fmaddsub_ps (_m256 a, _m256 b, _m256 c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2

## VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD – Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , subtract/add elements in <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , subtract/add elements in <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , subtract/add elements in <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , subtract/add elements in <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , subtract/add elements in <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , subtract/add elements in <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFMSUBADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFMSUBADD132PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192])
FI
```

### **VFMSUBADD213PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192])
FI
```

### **VFMSUBADD231PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192])
FI
```

## Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBADD132PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADD213PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADD231PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADD132PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);
VFMSUBADD213PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);
```

VFMSUBADD231PD: \_\_m256d \_mm256\_fmsubadd\_pd (\_\_m256d a, \_\_m256d b, \_\_m256d c);

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

See Exceptions Type 2

## VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS – Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , subtract/add elements in <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , subtract/add elements in <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , subtract/add elements in <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , subtract/add elements in <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , subtract/add elements in <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , subtract/add elements in <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFMSUBADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## Operation

In the operations below, "+" , "-" , and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFMSUBADD132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFMSUBADD213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFMSUBADD231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUBADD132PS: `_m128_mm_fmsubadd_ps (_m128 a, _m128 b, _m128 c);`  
VFMSUBADD213PS: `_m128_mm_fmsubadd_ps (_m128 a, _m128 b, _m128 c);`  
VFMSUBADD231PS: `_m128_mm_fmsubadd_ps (_m128 a, _m128 b, _m128 c);`  
VFMSUBADD132PS: `_m256_mm256_fmsubadd_ps (_m256 a, _m256 b, _m256 c);`  
VFMSUBADD213PS: `_m256_mm256_fmsubadd_ps (_m256 a, _m256 b, _m256 c);`  
VFMSUBADD231PS: `_m256_mm256_fmsubadd_ps (_m256 a, _m256 b, _m256 c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2

## VFMSUB132PD/VFMSUB213PD/VFMSUB231PD — Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9A /r VFMSUB132PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 AA /r VFMSUB213PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 BA /r VFMSUB231PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , subtract <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 9A /r VFMSUB132PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , subtract <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 AA /r VFMSUB213PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , subtract <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 BA /r VFMSUB231PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , subtract <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PD:** Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PD:** Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PD:** Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFMSUB132PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFMSUB213PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFMSUB231PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUB132PD: `__m128d_mm_fmsub_pd (__m128d a, __m128d b, __m128d c);`  
VFMSUB213PD: `__m128d_mm_fmsub_pd (__m128d a, __m128d b, __m128d c);`  
VFMSUB231PD: `__m128d_mm_fmsub_pd (__m128d a, __m128d b, __m128d c);`  
VFMSUB132PD: `__m256d_mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);`  
VFMSUB213PD: `__m256d_mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);`  
VFMSUB231PD: `__m256d_mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2

## VFMSUB132PS/VFMSUB213PS/VFMSUB231PS – Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9A /r VFMSUB132PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 AA /r VFMSUB213PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 BA /r VFMSUB231PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , subtract <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 9A /r VFMSUB132PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , subtract <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 AA /r VFMSUB213PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , subtract <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.0 BA /r VFMSUB231PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , subtract <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## Operation

In the operations below, "+" , "-" , and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFMSUB132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFMSUB213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFMSUB231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUB132PS: `__m128_mm_fmsub_ps (__m128 a, __m128 b, __m128 c);`  
VFMSUB213PS: `__m128_mm_fmsub_ps (__m128 a, __m128 b, __m128 c);`  
VFMSUB231PS: `__m128_mm_fmsub_ps (__m128 a, __m128 b, __m128 c);`  
VFMSUB132PS: `__m256_mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);`  
VFMSUB213PS: `__m256_mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);`  
VFMSUB231PS: `__m256_mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2

## VFMSUB132SD/VFMSUB213SD/VFMSUB231SD – Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9B /r VFMSUB132SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 AB /r VFMSUB213SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 BB /r VFMSUB231SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , subtract <i>xmm0</i> and put result in <i>xmm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm\_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

**VFMSUB132SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[63:0]*\text{SRC3}[63:0] - \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VFMSUB213SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[63:0]*\text{DEST}[63:0] - \text{SRC3}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VFMSUB231SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[63:0]*\text{SRC3}[63:0] - \text{DEST}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUB132SD: `__m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);`  
 VFMSUB213SD: `__m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);`  
 VFMSUB231SD: `__m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 3

## VFMSUB132SS/VFMSUB213SS/VFMSUB231SS – Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9B /r VFMSUB132SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 AB /r VFMSUB213SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 BB /r VFMSUB231SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , subtract <i>xmm0</i> and put result in <i>xmm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm\_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

**VFMSUB132SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[31:0]*\text{SRC3}[31:0] - \text{SRC2}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VFMSUB213SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[31:0]*\text{DEST}[31:0] - \text{SRC3}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VFMSUB231SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[31:0]*\text{SRC3}[63:0] - \text{DEST}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUB132SS: `__m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);`  
VFMSUB213SS: `__m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);`  
VFMSUB231SS: `__m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 3

## VFNMADD132PD/VFNMADD213PD/VFNMADD231PD – Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9C /r VFNMADD132PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 AC /r VFNMADD213PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 BC /r VFNMADD231PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 9C /r VFNMADD132PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , negate the multiplication result and add to <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 AC /r VFNMADD213PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , negate the multiplication result and add to <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 BC /r VFNMADD231PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , negate the multiplication result and add to <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFNMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a

## INSTRUCTION SET REFERENCE, N-Z

XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

#### VFNMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

#### VFNMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

#### VFNMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
```

```
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

### **Intel C/C++ Compiler Intrinsic Equivalent**

VFNMADD132PD: `__m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);`  
 VFNMADD213PD: `__m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);`  
 VFNMADD231PD: `__m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);`  
 VFNMADD132PD: `__m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);`  
 VFNMADD213PD: `__m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);`  
 VFNMADD231PD: `__m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);`

### **SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

### **Other Exceptions**

See Exceptions Type 2

## VFNMADD132PS/VFNMADD213PS/VFNMADD231PS – Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9C /r VFNMADD132PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 AC /r VFNMADD213PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 BC /r VFNMADD231PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 9C /r VFNMADD132PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , negate the multiplication result and add to <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 AC /r VFNMADD213PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , negate the multiplication result and add to <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.0 BC /r VFNMADD231PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , negate the multiplication result and add to <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFNMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## Operation

In the operations below, "+" , "-" , and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFNMADD132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL =8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFNMADD213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL =8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFNMADD231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL =8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0

```

Fl

**Intel C/C++ Compiler Intrinsic Equivalent**

VFNMADD132PS: `__m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);`  
VFNMADD213PS: `__m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);`  
VFNMADD231PS: `__m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);`  
VFNMADD132PS: `__m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);`  
VFNMADD213PS: `__m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);`  
VFNMADD231PS: `__m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2

## VFNMADD132SD/VFNMADD213SD/VFNMADD231SD – Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9D /r VFNMADD132SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 AD /r VFNMADD213SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 BD /r VFNMADD231SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm0</i> and put result in <i>xmm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFNMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm\_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

**VFNMADD132SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(- (\text{DEST}[63:0]*\text{SRC3}[63:0]) + \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VFNMADD213SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(- (\text{SRC2}[63:0]*\text{DEST}[63:0]) + \text{SRC3}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**VFNMADD231SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(- (\text{SRC2}[63:0]*\text{SRC3}[63:0]) + \text{DEST}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

VFNMADD132SD: `__m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);`  
VFNMADD213SD: `__m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);`  
VFNMADD231SD: `__m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 3

## VFNMADD132SS/VFNMADD213SS/VFNMADD231SS – Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9D /r VFNMADD132SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 AD /r VFNMADD213SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 BD /r VFNMADD231SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm0</i> and put result in <i>xmm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm\_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

**VFNMADD132SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(- (\text{DEST}[31:0]*\text{SRC3}[31:0]) + \text{SRC2}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VFNMADD213SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(- (\text{SRC2}[31:0]*\text{DEST}[31:0]) + \text{SRC3}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VFNMADD231SS DEST, SRC2, SRC3**

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl\_MXCSR}(- (\text{SRC2}[31:0]*\text{SRC3}[63:0]) + \text{DEST}[31:0])$   
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

VFNMADD132SS: `__m128_mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);`  
VFNMADD213SS: `__m128_mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);`  
VFNMADD231SS: `__m128_mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 3

## VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD – Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9E /r VFNMSUB132PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 AE /r VFNMSUB213PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 BE /r VFNMSUB231PD <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 9E /r VFNMSUB132PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , negate the multiplication result and subtract <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 AE /r VFNMSUB213PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , negate the multiplication result and subtract <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 BE /r VFNMSUB231PD <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , negate the multiplication result and subtract <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFNMSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a

XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFNMSUB132PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFNMSUB213PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFNMSUB231PD DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}

```

```
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

### **Intel C/C++ Compiler Intrinsic Equivalent**

```
VFNMSUB132PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUB213PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUB231PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUB132PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);
VFNMSUB213PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);
VFNMSUB231PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);
```

### **SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

### **Other Exceptions**

See Exceptions Type 2

## VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS – Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9E /r VFNMSUB132PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 AE /r VFNMSUB213PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 BE /r VFNMSUB231PS <i>xmm0, xmm1, xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 9E /r VFNMSUB132PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , negate the multiplication result and subtract <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 AE /r VFNMSUB213PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , negate the multiplication result and subtract <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.0 BE /r VFNMSUB231PS <i>ymm0, ymm1, ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , negate the multiplication result and subtract <i>ymm0</i> and put result in <i>ymm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvv (r)	ModRM:r/m (r)	NA

### Description

VFNMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a

XMM register or a 128-bit memory location and encoded in rm\_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

### **VFNMSUB132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFNMSUB213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

### **VFNMSUB231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}

```

```
IF (VEX.128) THEN  
    DEST[VLMAX-1:128] ← 0  
FI
```

### Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132PS: `_m128_mm_fnmsub_ps (_m128 a, _m128 b, _m128 c);`  
VFNMSUB213PS: `_m128_mm_fnmsub_ps (_m128 a, _m128 b, _m128 c);`  
VFNMSUB231PS: `_m128_mm_fnmsub_ps (_m128 a, _m128 b, _m128 c);`  
VFNMSUB132PS: `_m256_mm256_fnmsub_ps (_m256 a, _m256 b, _m256 c);`  
VFNMSUB213PS: `_m256_mm256_fnmsub_ps (_m256 a, _m256 b, _m256 c);`  
VFNMSUB231PS: `_m256_mm256_fnmsub_ps (_m256 a, _m256 b, _m256 c);`

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

See Exceptions Type 2

## VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD – Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9F /r VFNMSUB132SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 AF /r VFNMSUB213SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 BF /r VFNMSUB231SD <i>xmm0, xmm1, xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm0</i> and put result in <i>xmm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFNMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm\_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

**VFNMSUB132SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(-(\text{DEST}[63:0]*\text{SRC3}[63:0]) - \text{SRC2}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VFNMSUB213SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(-(\text{SRC2}[63:0]*\text{DEST}[63:0]) - \text{SRC3}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**VFNMSUB231SD DEST, SRC2, SRC3**

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(-(\text{SRC2}[63:0]*\text{SRC3}[63:0]) - \text{DEST}[63:0])$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$   
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

VFNMSUB132SD: `__m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);`  
VFNMSUB213SD: `__m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);`  
VFNMSUB231SD: `__m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 3

## VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS – Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9F /r VFNMSUB132SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 AF /r VFNMSUB213SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 BF /r VFNMSUB231SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm0</i> and put result in <i>xmm0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg\_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm\_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

### Operation

In the operations below, "+", "-", and "\*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

#### VFNMSUB132SS DEST, SRC2, SRC3

DEST[31:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (DEST[31:0]\*SRC3[31:0]) - SRC2[31:0])  
DEST[127:32]  $\leftarrow$  DEST[127:32]

## INSTRUCTION SET REFERENCE, N-Z

DEST[VLMAX-1:128]  $\leftarrow$  0  
**VFNMSUB213SS DEST, SRC2, SRC3**  
DEST[31:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (SRC2[31:0]\*DEST[31:0]) - SRC3[31:0])  
DEST[127:32]  $\leftarrow$  DEST[127:32]  
DEST[VLMAX-1:128]  $\leftarrow$  0

**VFNMSUB231SS DEST, SRC2, SRC3**  
DEST[31:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (SRC2[31:0]\*SRC3[63:0]) - DEST[31:0])  
DEST[127:32]  $\leftarrow$  DEST[127:32]  
DEST[VLMAX-1:128]  $\leftarrow$  0

### Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SS: `__m128_mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);`  
VFNMSUB213SS: `__m128_mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);`  
VFNMSUB231SS: `__m128_mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);`

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

See Exceptions Type 3

**VGATHERDPD/VGATHERQPD – Gather Packed DP FP Values Using Signed Dword/Qword Indices**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/3 2-bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
VEX.DDS.128.66.0F38.W1 92 /r VGATHERDPD <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W1 93 /r VGATHERQPD <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W1 92 /r VGATHERDPD <i>ymm1</i> , <i>vm32x</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W1 93 /r VGATHERQPD <i>ymm1</i> , <i>vm64y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather double-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w) BaseReg (R): VSIB:base, VectorReg(R): VSIB:index		VEX.vvvv (r, w)	NA

**Description**

The instruction conditionally loads up to 2 or 4 double-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double-precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double-precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double-precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST  $\leftarrow$  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  $\leftarrow$  SRC3;

### VGATHERDPD (VEX.128 version)

```

FOR j  $\leftarrow$  0 to 1
    i  $\leftarrow$  j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i]  $\leftarrow$  0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i]  $\leftarrow$  0;
    FI;
ENDFOR
FOR j  $\leftarrow$  0 to 1
    k  $\leftarrow$  j * 32;
    i  $\leftarrow$  j * 64;
    DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i]  $\leftarrow$  FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i]  $\leftarrow$  0;
ENDFOR

```

MASK[VLMAX-1:128]  $\leftarrow$  0;  
 DEST[VLMAX-1:128]  $\leftarrow$  0;  
 (non-masked elements of the mask register have the content of respective element cleared)

**VGATHERQPD (VEX.128 version)**

```
FOR j  $\leftarrow$  0 to 1
  i  $\leftarrow$  j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i]  $\leftarrow$  0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +63:i]  $\leftarrow$  0;
  FI;
ENDFOR
FOR j  $\leftarrow$  0 to 1
  i  $\leftarrow$  j * 64;
  DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i]  $\leftarrow$  FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
  FI;
  MASK[i +63:i]  $\leftarrow$  0;
ENDFOR
MASK[VLMAX-1:128]  $\leftarrow$  0;
DEST[VLMAX-1:128]  $\leftarrow$  0;
(non-masked elements of the mask register have the content of respective element cleared)
```

**VGATHERQPD (VEX.256 version)**

```
FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i]  $\leftarrow$  0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +63:i]  $\leftarrow$  0;
  FI;
ENDFOR
FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 64;
  DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i]  $\leftarrow$  FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63:i]  $\leftarrow$  0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)
```

**VGATHERDPD (VEX.256 version)**

```
FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i]  $\leftarrow$  0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +63:i]  $\leftarrow$  0;
  FI;
ENDFOR
FOR j  $\leftarrow$  0 to 3
```

```

k ← j * 32;
i ← j * 64;
DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
IF MASK[63+i] THEN
    DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VGATHERDPD: __m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);
VGATHERDPD: __m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);
VGATHERDPD: __m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);
VGATHERDPD: __m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);
VGATHERQPD: __m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);
VGATHERQPD: __m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);
VGATHERQPD: __m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);
VGATHERQPD: __m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 12

**VGATHERDPS/VGATHERQPS – Gather Packed SP FP values Using Signed Dword/Qword Indices**

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS <i>ymm1</i> , <i>vm32y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32y</i> , gather single-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64y</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w) BaseReg (R): VSIB:base, VectorReg(R): VSIB:index		VEX.vvvv (r, w)	NA

**Description**

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST  $\leftarrow$  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  $\leftarrow$  SRC3;

### VGATHERDPS (VEX.128 version)

```

FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i]  $\leftarrow$  0xFFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +31:i]  $\leftarrow$  0;
  FI;
ENDFOR
MASK[VLMAX-1:128]  $\leftarrow$  0;
FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 32;
  DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i]  $\leftarrow$  FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i]  $\leftarrow$  0;
ENDFOR
  
```

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0;$   
 (non-masked elements of the mask register have the content of respective element cleared)

**VGATHERQPS (VEX.128 version)**

```

FOR j← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
MASK[VLMAX-1:128] ← 0;
FOR j← 0 to 1
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[VLMAX-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
  
```

**VGATHERRDPS (VEX.256 version)**

```

FOR j← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
FOR j← 0 to 7
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)
  
```

**VGATHERQPS (VEX.256 version)**

```

FOR j← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
  
```

```

FOR j<- 0 to 3
    k <- j * 64;
    i <- j * 32;
    DATA_ADDR <- BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] <- FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] <- 0;
ENDFOR
MASK[VLMAX-1:128] <- 0;
DEST[VLMAX-1:128] <- 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

### Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPS:	<code>__m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);</code>
VGATHERDPS:	<code>__m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);</code>
VGATHERDPS:	<code>__m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);</code>
VGATHERDPS:	<code>__m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);</code>
VGATHERQPS:	<code>__m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);</code>
VGATHERQPS:	<code>__m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);</code>
VGATHERQPS:	<code>__m128 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);</code>
VGATHERQPS:	<code>__m128 _mm256_mask_i64gather_ps (__m128 src, float const * base, __m256i index, __m128 mask, const int scale);</code>

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 12

## VPGATHERDD/VPGATHERQD – Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD <i>xmm1, vm32x, xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W0 91 /r VPGATHERQD <i>xmm1, vm64x, xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W0 90 /r VPGATHERDD <i>ymm1, vm32y, ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32y</i> , gather dword from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W0 91 /r VPGATHERQD <i>xmm1, vm64y, xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

### Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 dword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction 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  $\leftarrow$  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  $\leftarrow$  SRC3;

### VPGATHERDD (VEX.128 version)

```

FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i]  $\leftarrow$  0xFFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +31:i]  $\leftarrow$  0;
  FI;
ENDFOR
MASK[VLMAX-1:128]  $\leftarrow$  0;
FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 32;
  DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i]  $\leftarrow$  FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i]  $\leftarrow$  0;
ENDFOR
DEST[VLMAX-1:128]  $\leftarrow$  0;
(non-masked elements of the mask register have the content of respective element cleared)
  
```

**VPGATHERQD (VEX.128 version)**

```

FOR j<- 0 to 3
    i <- j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] <- 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] <- 0;
    FI;
ENDFOR
MASK[VLMAX-1:128] <- 0;
FOR j<- 0 to 1
    k <- j * 64;
    i <- j * 32;
    DATA_ADDR <- BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] <- FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] <- 0;
ENDFOR
MASK[127:64] <- 0;
DEST[VLMAX-1:64] <- 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

**VPGATHERDD (VEX.256 version)**

```

FOR j<- 0 to 7
    i <- j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] <- 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] <- 0;
    FI;
ENDFOR
FOR j<- 0 to 7
    i <- j * 32;
    DATA_ADDR <- BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] <- FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] <- 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

**VPGATHERQD (VEX.256 version)**

```

FOR j<- 0 to 7
    i <- j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] <- 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] <- 0;
    FI;
ENDFOR
FOR j<- 0 to 3
    k <- j * 64;
    i <- j * 32;

```

## INSTRUCTION SET REFERENCE, N-Z

```
DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +31:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERDD: __m256i _mm256_i32gather_epi32 ( int const * base, __m256i index, const int scale);
VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);
VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQD: __m128i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);
VPGATHERQD: __m128i _mm256_mask_i64gather_epi32 (__m128i src, int const * base, __m256i index, __m128i mask, const int scale);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 12

## VPGATHERDQ/VPGATHERQQ – Gather Packed Qword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 90 /r VPGATHERDQ <i>xmm1, vm32x, xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather qword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W1 91 /r VPGATHERQQ <i>xmm1, vm64x, xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather qword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W1 90 /r VPGATHERDQ <i>ymm1, vm32x, ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather qword values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W1 91 /r VPGATHERQQ <i>ymm1, vm64y, ymm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather qword values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

### Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST  $\leftarrow$  SRC1;  
 BASE\_ADDR: base register encoded in VSIB addressing;  
 VINTEX: the vector index register encoded by VSIB addressing;  
 SCALE: scale factor encoded by SIB:[7:6];  
 DISP: optional 1, 4 byte displacement;  
 MASK  $\leftarrow$  SRC3;

### VPGATHERDQ (VEX.128 version)

```

FOR j  $\leftarrow$  0 to 1
  i  $\leftarrow$  j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i]  $\leftarrow$  0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
  ELSE
    MASK[i +63:i]  $\leftarrow$  0;
  FI;
ENDFOR
FOR j  $\leftarrow$  0 to 1
  k  $\leftarrow$  j * 32;
  i  $\leftarrow$  j * 64;
  DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i]  $\leftarrow$  FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63:i]  $\leftarrow$  0;
ENDFOR
MASK[VLMAX-1:128]  $\leftarrow$  0;
DEST[VLMAX-1:128]  $\leftarrow$  0;
  
```

(non-masked elements of the mask register have the content of respective element cleared)

#### **VPGATHERQQ (VEX.128 version)**

```

FOR j<- 0 to 1
    i <- j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] <- 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] <- 0;
    FI;
ENDFOR
FOR j<- 0 to 1
    i <- j * 64;
    DATA_ADDR <- BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] <- FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] <- 0;
ENDFOR
MASK[VLMAX-1:128] <- 0;
DEST[VLMAX-1:128] <- 0;

```

(non-masked elements of the mask register have the content of respective element cleared)

#### **VPGATHERQQ (VEX.256 version)**

```

FOR j<- 0 to 3
    i <- j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] <- 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] <- 0;
    FI;
ENDFOR
FOR j<- 0 to 3
    i <- j * 64;
    DATA_ADDR <- BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] <- FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] <- 0;
ENDFOR

```

(non-masked elements of the mask register have the content of respective element cleared)

#### **VPGATHERDQ (VEX.256 version)**

```

FOR j<- 0 to 3
    i <- j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] <- 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] <- 0;
    FI;
ENDFOR
FOR j<- 0 to 3
    k <- j * 32;
    i <- j * 64;

```

```

DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
IF MASK[63+i] THEN
    DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPGATHERDQ: __m128i _mm_i32gather_epi64 (int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, int64 const * base, __m128i index, __m128i mask, const int scale);
VPGATHERDQ: __m256i _mm256_i32gather_epi64 ( int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, int64 const * base, __m128i index, __m256i mask, const int scale);
VPGATHERQQ: __m128i _mm_i64gather_epi64 (int64 const * base, __m128i index, const int scale);
VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, int64 const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQQ: __m256i _mm256_i64gather_epi64 (int64 const * base, __m256i index, const int scale);
VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, int64 const * base, __m256i index, __m256i mask, const int scale);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 12

## VINSETRF128 – Insert Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSETRF128 <i>ymm1, ymm2, xmm3/m128, imm8</i>	RVM	V/V	AVX	Insert a single precision floating-point value selected by <i>imm8</i> from <i>xmm3/m128</i> into <i>ymm2</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>ymm1</i> as indicated in <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an insertion of 128-bits of packed floating-point values from the second source operand (third operand) into an the destination operand (first operand) at an 128-bit offset from *imm8[0]*. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

### Operation

```

TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] ← SRC2[127:0]
    1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ← TEMP

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

INSERTF128:   __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);
INSERTF128:   __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);
INSERTF128:   __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 6; additionally

#UD                    If VEX.W = 1.

**VINSERTI128 – Insert Packed Integer Values**

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI128 <i>ymm1, ymm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX2	Insert 128-bits of integer data from <i>xmm3/mem</i> and the remaining values from <i>ymm2</i> into <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

**Description**

Performs an insertion of 128-bits of packed integer data from the second source operand (third operand) into an the destination operand (first operand) at a 128-bit offset from imm8[0]. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

VEX.L must be 1; an attempt to execute this instruction with VEX.L=0 will cause #UD.

**Operation****VINSERTI128**

```
TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] ← SRC2[127:0]
    1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ← TEMP
```

**Intel C/C++ Compiler Intrinsic Equivalent**

VINSERTI128: \_\_m256i \_mm256\_inserti128\_si256 (\_\_m256i a, \_\_m128i b, int offset);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 6; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

## VMASKMOV—Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 2C /r VMASKMOVPS <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX	Conditionally load packed single-precision values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2C /r VMASKMOVPS <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX	Conditionally load packed single-precision values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2D /r VMASKMOVPD <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX	Conditionally load packed double-precision values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2D /r VMASKMOVPD <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX	Conditionally load packed double-precision values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2E /r VMASKMOVPS <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX	Conditionally store packed single-precision values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2E /r VMASKMOVPS <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX	Conditionally store packed single-precision values from <i>ymm2</i> using mask in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2F /r VMASKMOVPD <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX	Conditionally store packed double-precision values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2F /r VMASKMOVPD <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX	Conditionally store packed double-precision values from <i>ymm2</i> using mask in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

### Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm\_field, and the destination register is encoded in reg\_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg\_field, and the destination memory location is encoded in rm\_field.

## Operation

### **VMASKMOVPS - 128-bit load**

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX-1:128] ← 0
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

### **VMASKMOVPD - 128-bit load**

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX-1:128] ← 0
```

### **VMASKMOVPD - 256-bit load**

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

### **VMASKMOVPS - 128-bit store**

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
```

### **VMASKMOVPS - 256-bit store**

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ← SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
```

### **VMASKMOVPD - 128-bit store**

```
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]
```

**VMASKMOVPD - 256-bit store**

IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]  
 IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]

**VMASKMOVPS - 256-bit load**

IF (SRC1[191]) DEST[191:128] ← SRC2[191:128]  
 IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

**Intel C/C++ Compiler Intrinsic Equivalent**

```
_m256 _mm256_maskload_ps(float const *a, __m256i mask)
void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)
__m256d _mm256_maskload_pd(double *a, __m256i mask);
void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);

__m128 _mm128_maskload_ps(float const *a, __m128i mask)
void _mm128_maskstore_ps(float *a, __m128i mask, __m128 b)
__m128d _mm128_maskload_pd(double *a, __m128i mask);
void _mm128_maskstore_pd(double *a, __m128i mask, __m128d b);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 6 (No AC# reported for any mask bit combinations);  
 additionally

#UD                  If VEX.W = 1.

**VPBLEND<sub>D</sub> – Blend Packed Dwords**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 -bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
VEX.NDS.128.66.0F3A.W0 02 /r ib VPBLEND <sub>D</sub> <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX2	Select dwords from <i>xmm2</i> and <i>xmm3/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.W0 02 /r ib VPBLEND <sub>D</sub> <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Select dwords from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in <i>imm8</i> and store the values into <i>ymm1</i> .

**Instruction Operand Encoding**

<b>Op/En</b>	<b>Operand 1</b>	<b>Operand 2</b>	<b>Operand 3</b>	<b>Operand 4</b>
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

**Description**

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**Operation****VPBLEND<sub>D</sub> (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]
```

**VPBLEND<sub>D</sub>D (VEX.128 encoded version)**

```

IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
DEST[VLMAX-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

**VPBLEND<sub>D</sub>D:**    `__m128i_mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)`  
**VPBLEND<sub>D</sub>D:**    `__m256i_mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                  If VEX.W = 1.

**VPBROADCAST—Broadcast Integer Data**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 -bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
VEX.128.66.0F38.W0 78 /r VPBROADCASTB <i>xmm1, xmm2/m8</i>	RM	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 78 /r VPBROADCASTB <i>ymm1, xmm2/m8</i>	RM	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in <i>ymm1</i> .
VEX.128.66.0F38.W0 79 /r VPBROADCASTW <i>xmm1, xmm2/m16</i>	RM	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 79 /r VPBROADCASTW <i>ymm1, xmm2/m16</i>	RM	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in <i>ymm1</i> .
VEX.128.66.0F38.W0 58 /r VPBROADCASTD <i>xmm1, xmm2/m32</i>	RM	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 58 /r VPBROADCASTD <i>ymm1, xmm2/m32</i>	RM	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in <i>ymm1</i> .
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ <i>xmm1, xmm2/m64</i>	RM	V/V	AVX2	Broadcast a qword element in mem to two locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ <i>ymm1, xmm2/m64</i>	RM	V/V	AVX2	Broadcast a qword element in mem to four locations in <i>ymm1</i> .
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 <i>ymm1, m128</i>	RM	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Load integer data from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

The destination operand is a YMM register. The source operand is 8-bit, 16-bit 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. VPBROADCASTB/D/W/Q also support XMM register as the source operand.

VBROADCASTI128: The destination operand is a YMM register. The source operand is 128-bit memory location. Register source encodings for VBROADCASTI128 are reserved and will #UD.

VPBROADCASTB/W/D/Q is supported in both 128-bit and 256-bit wide versions.

VBROADCASTI128 is only supported as a 256-bit wide version.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. Attempts to execute any VPBROADCAST\* instruction with VEX.W = 1 will cause #UD. If VBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

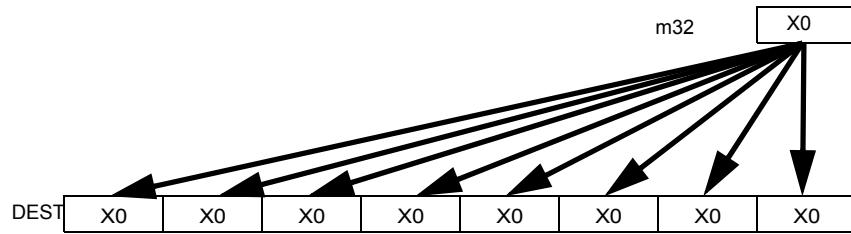


Figure 4-33. VPBROADCASTD Operation (VEX.256 encoded version)

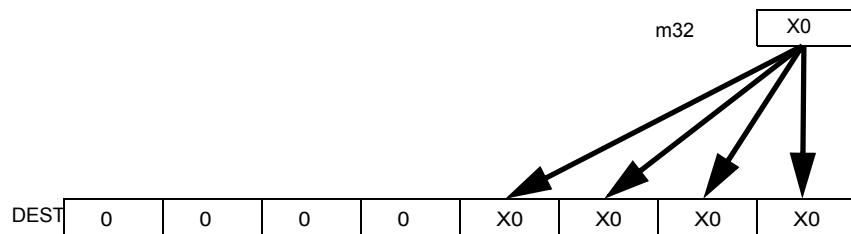


Figure 4-34. VPBROADCASTD Operation (128-bit version)

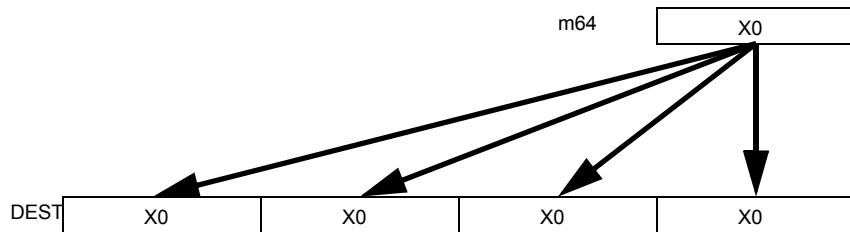


Figure 4-35. VPBROADCASTQ Operation

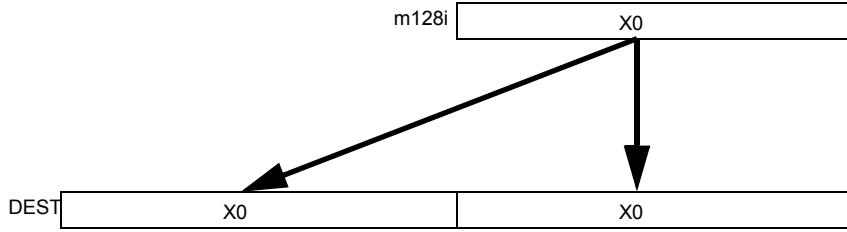


Figure 4-36. VBROADCASTI128 Operation

## Operation

### VPBROADCASTB (VEX.128 encoded version)

```
temp <- SRC[7:0]
FOR j <- 0 TO 15
  DEST[7+j*8: j*8] <- temp
ENDFOR
DEST[VLMAX-1:128] <- 0
```

### VPBROADCASTB (VEX.256 encoded version)

```
temp <- SRC[7:0]
FOR j <- 0 TO 31
  DEST[7+j*8: j*8] <- temp
ENDFOR
```

### VPBROADCASTW (VEX.128 encoded version)

```
temp <- SRC[15:0]
FOR j <- 0 TO 7
  DEST[15+j*16: j*16] <- temp
ENDFOR
DEST[VLMAX-1:128] <- 0
```

### VPBROADCASTW (VEX.256 encoded version)

```
temp <- SRC[15:0]
FOR j <- 0 TO 15
  DEST[15+j*16: j*16] <- temp
ENDFOR
```

### VPBROADCASTD (128 bit version)

```
temp <- SRC[31:0]
FOR j <- 0 TO 3
  DEST[31+j*32: j*32] <- temp
ENDFOR
DEST[VLMAX-1:128] <- 0
```

**VPBROADCASTD (VEX.256 encoded version)**

```
temp ← SRC[31:0]
FOR j ← 0 TO 7
  DEST[31+j*32:j*32] ← temp
ENDFOR
```

**VPBROADCASTQ (VEX.128 encoded version)**

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[VLMAX-1:128] ← 0
```

**VPBROADCASTQ (VEX.256 encoded version)**

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
```

**VBROADCASTI128**

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[VLMAX-1:128] ← temp
```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPBROADCASTB:	<code>_m256i _mm256_broadcastb_epi8(__m128i );</code>
VPBROADCASTW:	<code>_m256i _mm256_broadcastw_epi16(__m128i );</code>
VPBROADCASTD:	<code>_m256i _mm256_broadcastd_epi32(__m128i );</code>
VPBROADCASTQ:	<code>_m256i _mm256_broadcastq_epi64(__m128i );</code>
VPBROADCASTB:	<code>_m128i _mm_broadcastb_epi8(__m128i );</code>
VPBROADCASTW:	<code>_m128i _mm_broadcastw_epi16(__m128i );</code>
VPBROADCASTD:	<code>_m128i _mm_broadcastd_epi32(__m128i );</code>
VPBROADCASTQ:	<code>_m128i _mm_broadcastq_epi64(__m128i );</code>
VBROADCASTI128:	<code>_m256i _mm256_broadcastsi128_si256(__m128i );</code>

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 6; additionally

#UD	If VEX.W = 1, If VEX.L = 0 for VBROADCASTI128.
-----	---

**VPERMD — Full Doublewords Element Permutation**

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 36 /r VPERMD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Permute doublewords in <i>ymm3/m256</i> using indexes in <i>ymm2</i> and store the result in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

**Description**

Use the index values in each dword element of the first source operand (the second operand) to select a dword element in the second source operand (the third operand), the resultant dword value from the second source operand is copied to the destination operand (the first operand) in the corresponding position of the index element. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

An attempt to execute VPERMD encoded with VEX.L= 0 will cause an #UD exception.

**Operation****VPERMD (VEX.256 encoded version)**

```

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMD: \_\_m256i \_mm256\_permutevar8x32\_epi32(\_\_m256i a, \_\_m256i offsets);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.L = 0 for VPERMD,  
                       If VEX.W = 1.

## VPERMPD — Permute Double-Precision Floating-Point Elements

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib VPERMPD <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Permute double-precision floating-point elements in <i>ymm2/m256</i> using indexes in <i>imm8</i> and store the result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Use two-bit index values in the immediate byte to select a double-precision floating-point element in the source operand; the resultant data from the source operand is copied to the corresponding element of the destination operand in the order of the index field. Note that this instruction permits a qword in the source operand to be copied to multiple location in the destination operand.

An attempt to execute VPERMPD encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### VPERMPD (VEX.256 encoded version)

```
DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPERMPD: `__m256d _mm256_permute4x64_pd(__m256d a, int control);`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 0.

## VPERMPS – Permute Single-Precision Floating-Point Elements

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 16 /r VPERMPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Permute single-precision floating-point elements in <i>ymm3/m256</i> using indexes in <i>ymm2</i> and store the result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Use the index values in each dword element of the first source operand (the second operand) to select a single-precision floating-point element in the second source operand (the third operand), the resultant data from the second source operand is copied to the destination operand (the first operand) in the corresponding position of the index element. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

An attempt to execute VPERMPS encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### VPERMPS (VEX.256 encoded version)

```

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];

```

### Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS: \_\_m256i \_mm256\_permutevar8x32\_ps(\_\_m256 a, \_\_m256i offsets)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4; additionally

#UD                  If VEX.L = 0,  
                        If VEX.W = 1.

## VPERMQ – Qwords Element Permutation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib VPERMQ <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Permute qwords in <i>ymm2/m256</i> using indexes in <i>imm8</i> and store the result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Use two-bit index values in the immediate byte to select a qword element in the source operand, the resultant qword value from the source operand is copied to the corresponding element of the destination operand in the order of the index field. Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

An attempt to execute VPERMQ encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### VPERMQ (VEX.256 encoded version)

```
DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPERMQ: \_\_m256i \_mm256\_permute4x64\_epi64(\_\_m256i a, int control)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4; additionally

#UD                    If VEX.L = 0.

**VPERM2I128 – Permute Integer Values**

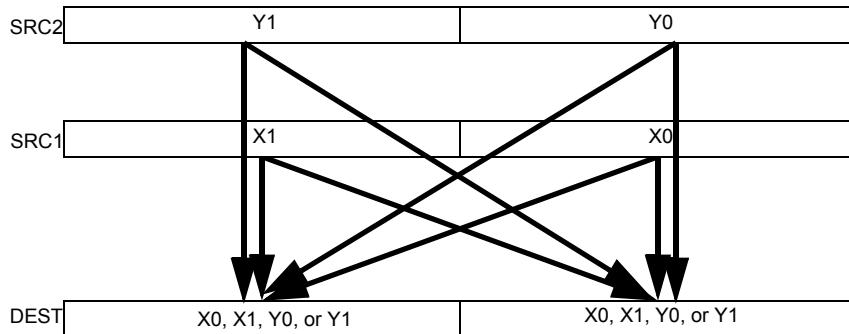
Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 46 /r ib VPERM2I128 <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Permute 128-bit integer data in <i>ymm2</i> and <i>ymm3/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

**Description**

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

**Figure 4-37. VPERM2I128 Operation**

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.

## Operation

### VPERM2I128

```
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC
CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[255:128] ← 0
FI
```

## Intel C/C++ Compiler Intrinsic Equivalent

`VPERM2I128: __m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)`

## SIMD Floating-Point Exceptions

None

## Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

**VPERMILPD — Permute Double-Precision Floating-Point Values**

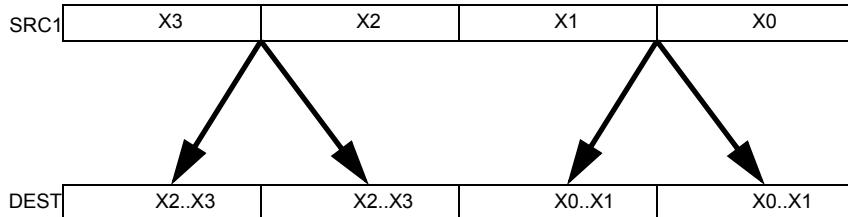
Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Permute double-precision floating-point values in <i>xmm2</i> using controls from <i>xmm3/mem</i> and store result in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Permute double-precision floating-point values in <i>ymm2</i> using controls from <i>ymm3/mem</i> and store result in <i>ymm1</i> .
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Permute double-precision floating-point values in <i>xmm2/mem</i> using controls from <i>imm8</i> .
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX	Permute double-precision floating-point values in <i>ymm2/mem</i> using controls from <i>imm8</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

**Description**

Permute double-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of the second source operand (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

**Figure 4-38. VPERMILPD operation**

There is one control byte per destination double-precision element. Each control byte is aligned with the low 8 bits of the corresponding double-precision destination element. Each control byte contains a 1-bit select field (see Figure 4-39) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

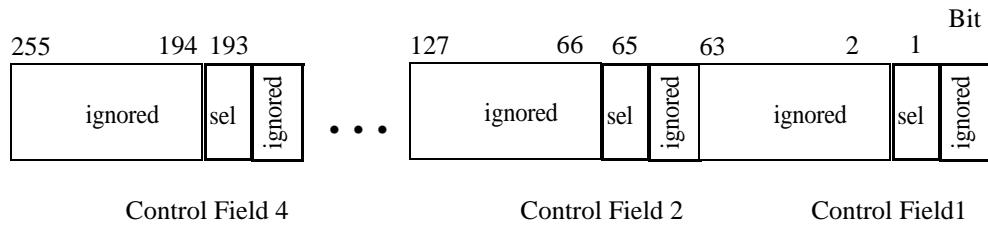


Figure 4-39. VPERMILPD Shuffle Control

(immediate control version)

Permute double-precision floating-point values in the first source operand (second operand) using two, 1-bit control fields in the low 2 bits of the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register.

Note: For the VEX.128.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

## Operation

### VPERMILPD (256-bit immediate version)

```

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
IF (imm8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]
IF (imm8[2] = 1) THEN DEST[191:128] ← SRC1[255:192]
IF (imm8[3] = 0) THEN DEST[255:192] ← SRC1[191:128]
IF (imm8[3] = 1) THEN DEST[255:192] ← SRC1[255:192]

```

### VPERMILPD (128-bit immediate version)

```

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

```

### VPERMILPD (256-bit variable version)

```

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128] ← SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128] ← SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192] ← SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192] ← SRC1[255:192]

```

**VPERMILPD (128-bit variable version)**

```

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMILPD:    __m128d _mm_permute_pd (__m128d a, int control)
VPERMILPD:    __m256d _mm256_permute_pd (__m256d a, int control)
VPERMILPD:    __m128d _mm_permutevar_pd (__m128d a, __m128i control);
VPERMILPD:    __m256d _mm256_permutevar_pd (__m256d a, __m256i control);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 6; additionally

#UD                  If VEX.W = 1

## VPERMILPS – Permute Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Permute single-precision floating-point values in <i>xmm2</i> using controls from <i>xmm3/mem</i> and store result in <i>xmm1</i> .
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Permute single-precision floating-point values in <i>xmm2/mem</i> using controls from <i>imm8</i> and store result in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Permute single-precision floating-point values in <i>ymm2</i> using controls from <i>ymm3/mem</i> and store result in <i>ymm1</i> .
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX	Permute single-precision floating-point values in <i>ymm2/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

(variable control version)

Permute single-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of corresponding elements the shuffle control (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

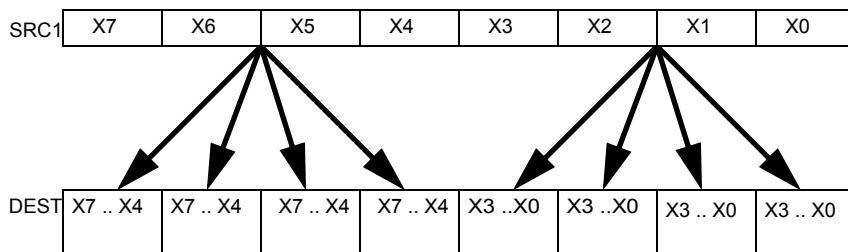
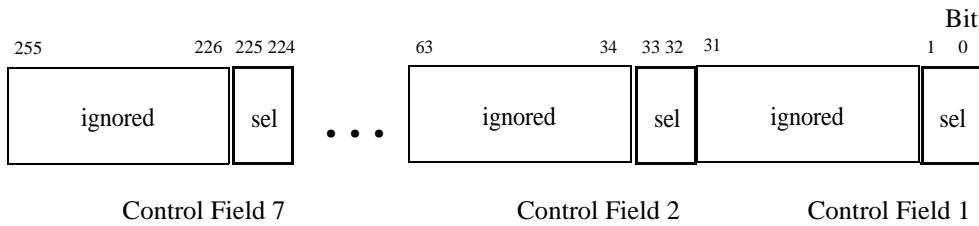


Figure 4-40. VPERMILPS Operation

There is one control byte per destination single-precision element. Each control byte is aligned with the low 8 bits of the corresponding single-precision destination element. Each control byte contains a 2-bit select field (see Figure 4-41) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.



**Figure 4-41.** VPERMILPS Shuffle Control

(immediate control version)

Permute single-precision floating-point values in the first source operand (second operand) using four 2-bit control fields in the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register. This is similar to a wider version of PSHUFD, just operating on single-precision floating-point values.

Note: For the VEX.128.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

## Operation

```

Select4(SRC, control) {
CASE (control[1:0]) OF
  0: TMP ← SRC[31:0];
  1: TMP ← SRC[63:32];
  2: TMP ← SRC[95:64];
  3: TMP ← SRC[127:96];
ESAC;
RETURN TMP
}

```

## VPERMILPS (256-bit immediate version)

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);

```

## VPERMILPS (128-bit immediate version)

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[VLMAX-1:128] ← 0

```

**VPERMILPS (256-bit variable version)**

```

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] ← Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] ← Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] ← Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] ← Select4(SRC1[255:128], SRC2[225:224]);

```

**VPERMILPS (128-bit variable version)**

```

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[VLMAX-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERM1LPS:      __m128 _mm_permute_ps (__m128 a, int control);
VPERM1LPS:      __m256 _mm256_permute_ps (__m256 a, int control);
VPERM1LPS:      __m128 _mm_permutevar_ps (__m128 a, __m128i control);
VPERM1LPS:      __m256 _mm256_permutevar_ps (__m256 a, __m256i control);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 6; additionally

#UD                  If VEX.W = 1.

**VPERM2F128 – Permute Floating-Point Values**

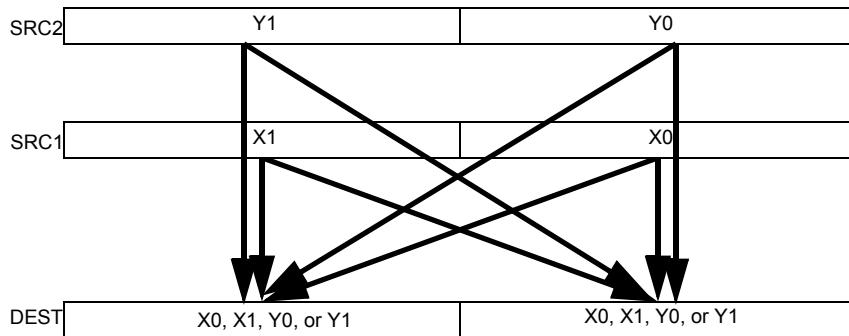
Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 06 /r ib VPERM2F128 <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Permute 128-bit floating-point fields in <i>ymm2</i> and <i>ymm3/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

**Description**

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

**Figure 4-42. VPERM2F128 Operation**

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.

## Operation

### VPERM2F128

```
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC
```

```
CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI
```

```
IF (imm8[7])
DEST[VLMAX-1:128] ← 0
FI
```

## Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128:	<code>_m256 _mm256_permute2f128_ps (_m256 a, _m256 b, int control)</code>
VPERM2F128:	<code>_m256d _mm256_permute2f128_pd (_m256d a, _m256d b, int control)</code>
VPERM2F128:	<code>_m256i _mm256_permute2f128_si256 (_m256i a, _m256i b, int control)</code>

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0
	If VEX.W = 1.

**VPMASKMOV – Conditional SIMD Integer Packed Loads and Stores**

<b>Opcode/ Instruction</b>	<b>Op/ En</b>	<b>64/32 -bit Mode</b>	<b>CPUID Feature Flag</b>	<b>Description</b>
VEX.NDS.128.66.0F38.W0 8C /r VPMASKMOVD <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX2	Conditionally load dword values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 8C /r VPMASKMOVD <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX2	Conditionally load dword values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W1 8C /r VPMASKMOVQ <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX2	Conditionally load qword values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W1 8C /r VPMASKMOVQ <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX2	Conditionally load qword values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 8E /r VPMASKMOVD <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX2	Conditionally store dword values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 8E /r VPMASKMOVD <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX2	Conditionally store dword values from <i>ymm2</i> using mask in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W1 8E /r VPMASKMOVQ <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX2	Conditionally store qword values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W1 8E /r VPMASKMOVQ <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX2	Conditionally store qword values from <i>ymm2</i> using mask in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv	ModRM:reg (r)	NA

**Description**

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm\_field, and the destination register is encoded in reg\_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg\_field, and the destination memory location is encoded in rm\_field.

## Operation

### VPMASKMOVD - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

### VPMASKMOVD - 128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX-1:128] ← 0
```

### VPMASKMOVQ - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

### VPMASKMOVQ - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX-1:128] ← 0
```

### VPMASKMOVD - 256-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ← SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
```

**VPMASKMOVD - 128-bit store**

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
```

**VPMASKMOVQ - 256-bit store**

```
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]
IF (SRC1[191]) DEST[191:128] ← SRC2[191:128]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]
```

**VPMASKMOVQ - 128-bit store**

```
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)
VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)
VPMASKMOVQ: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);
VPMASKMOVQ: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);
VPMASKMOVD: __m128i _mm_maskload_epi32(int const *a, __m128i mask)
VPMASKMOVD: void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)
VPMASKMOVQ: __m128i _mm_maskload_epi64(__int cont *a, __m128i mask);
VPMASKMOVQ: void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128 b);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 6 (No AC# reported for any mask bit combinations).

## VPSLLVD/VPSLLVQ – Variable Bit Shift Left Logical

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 47 /r VPSLLVD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>xmm2</i> left by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in 0s.
VEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in quadwords in <i>xmm2</i> left by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in 0s.
VEX.NDS.256.66.0F38.W0 47 /r VPSLLVD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>ymm2</i> left by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in 0s.
VEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in quadwords in <i>ymm2</i> left by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in 0s.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Shifts the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

### Operation

#### VPSLLVD (VEX.128 version)

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ← ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX-1:128] ← 0;
```

**VPSLLVD (VEX.256 version)**

```
COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] ← 0;
```

**VPSLLVQ (VEX.128 version)**

```
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX-1:128] ← 0;
```

**VPSLLVQ (VEX.256 version)**

```
COUNT_0 ← SRC2[5 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[197 : 192];
IF COUNT_0 < 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;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSLLVD: \_\_m256i \_mm256\_sllv\_epi32 (\_\_m256i m, \_\_m256i count)  
VPSLLVD: \_\_m128i \_mm\_sllv\_epi32 (\_\_m128i m, \_\_m128i count)  
VPSLLVQ: \_\_m256i \_mm256\_sllv\_epi64 (\_\_m256i m, \_\_m256i count)  
VPSLLVQ: \_\_m128i \_mm\_sllv\_epi64 (\_\_m128i m, \_\_m128i count)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4

## VPSRAVD – Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 46 /r VPSRAVD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>xmm2</i> right by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in the sign bits.
VEX.NDS.256.66.0F38.W0 46 /r VPSRAVD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>ymm2</i> right by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in the sign bits.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Shifts the bits in the individual doubleword data elements in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in each data element are shifted right, the empty high-order bits are filled with the sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31, then the destination data element are filled with the corresponding sign bit of the source element.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

### Operation

#### VPSRAVD (VEX.128 version)

```

COUNT_0 ← SRC2[31:0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127:112];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
ELSE
    For (i = 0 to 31) DEST[i + 0] ← (SRC1[31]);
FI;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
    DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
ELSE
    For (i = 0 to 31) DEST[i + 96] ← (SRC1[127]);
FI;
DEST[VLMAX-1:128] ← 0;
```

**VPSRAVD (VEX.256 version)**

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
ELSE
    For (i = 0 to 31) DEST[i + 0] ← (SRC1[31]);
FI;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
ELSE
    For (i = 0 to 31) DEST[i + 224] ← (SRC1[255]);
FI;

```

**Intel C/C++ Compiler Intrinsic Equivalent**VPSRAVD: `__m256i _mm256_srav_epi32 (__m256i m, __m256i count)`VPSRAVD: `__m128i _mm_srav_epi32 (__m128i m, __m128i count)`**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                    If VEX.W = 1.

## VPSRLVD/VPSRLVQ – Variable Bit Shift Right Logical

Opcode/ Instruction	Op/ EN	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>xmm2</i> right by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in 0s.
VEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in quadwords in <i>xmm2</i> right by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in 0s.
VEX.NDS.256.66.0F38.W0 45 /r VPSRLVD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>ymm2</i> right by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in 0s.
VEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in quadwords in <i>ymm2</i> right by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in 0s.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Shifts the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

### Operation

#### VPSRLVD (VEX.128 version)

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ← ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX-1:128] ← 0;
```

**VPSRLVD (VEX.256 version)**

```
COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
DEST[255:224] ← 0;
```

**VPSRLVQ (VEX.128 version)**

```
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
DEST[127:64] ← 0;
DEST[VLMAX-1:128] ← 0;
```

**VPSRLVQ (VEX.256 version)**

```
COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
DEST[255:192] ← 0;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRLVD: \_\_m256i \_mm256\_srlv\_epi32 (\_\_m256i m, \_\_m256i count);  
VPSRLVD: \_\_m128i \_mm\_srlv\_epi32 (\_\_m128i m, \_\_m128i count);  
VPSRLVQ: \_\_m256i \_mm256\_srlv\_epi64 (\_\_m256i m, \_\_m256i count);  
VPSRLVQ: \_\_m128i \_mm\_srlv\_epi64 (\_\_m128i m, \_\_m128i count);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4

## VTESTPD/VTESTPS—Packed Bit Test

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0E /r VTESTPS <i>xmm1</i> , <i>xmm2/m128</i>	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 <i>ymm1</i> , <i>ymm2/m256</i>	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 <i>xmm1</i> , <i>xmm2/m128</i>	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 <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

### Description

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND 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.

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)

$\text{TEMP}[127:0] \leftarrow \text{SRC}[127:0] \text{ AND } \text{DEST}[127:0]$   
IF ( $\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = 0$ )  
THEN ZF  $\leftarrow 1$ ;  
ELSE ZF  $\leftarrow 0$ ;

$\text{TEMP}[127:0] \leftarrow \text{SRC}[127:0] \text{ AND NOT } \text{DEST}[127:0]$   
IF ( $\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = 0$ )  
THEN CF  $\leftarrow 1$ ;  
ELSE CF  $\leftarrow 0$ ;  
DEST (unmodified)  
AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow 0$ ;

### VTESTPS (VEX.256 encoded version)

$\text{TEMP}[255:0] \leftarrow \text{SRC}[255:0] \text{ AND } \text{DEST}[255:0]$   
IF ( $\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = \text{TEMP}[160] = \text{TEMP}[191] = \text{TEMP}[224] = \text{TEMP}[255] = 0$ )  
THEN ZF  $\leftarrow 1$ ;  
ELSE ZF  $\leftarrow 0$ ;

$\text{TEMP}[255:0] \leftarrow \text{SRC}[255:0] \text{ AND NOT } \text{DEST}[255:0]$   
IF ( $\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = \text{TEMP}[160] = \text{TEMP}[191] = \text{TEMP}[224] = \text{TEMP}[255] = 0$ )  
THEN CF  $\leftarrow 1$ ;  
ELSE CF  $\leftarrow 0$ ;  
DEST (unmodified)  
AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow 0$ ;

### VTESTPD (128-bit version)

$\text{TEMP}[127:0] \leftarrow \text{SRC}[127:0] \text{ AND } \text{DEST}[127:0]$   
IF ( $\text{TEMP}[63] = \text{TEMP}[127] = 0$ )  
THEN ZF  $\leftarrow 1$ ;  
ELSE ZF  $\leftarrow 0$ ;

$\text{TEMP}[127:0] \leftarrow \text{SRC}[127:0] \text{ AND NOT } \text{DEST}[127:0]$   
IF ( $\text{TEMP}[63] = \text{TEMP}[127] = 0$ )  
THEN CF  $\leftarrow 1$ ;  
ELSE CF  $\leftarrow 0$ ;  
DEST (unmodified)  
AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow 0$ ;

### VTESTPD (VEX.256 encoded version)

$\text{TEMP}[255:0] \leftarrow \text{SRC}[255:0] \text{ AND } \text{DEST}[255:0]$   
IF ( $\text{TEMP}[63] = \text{TEMP}[127] = \text{TEMP}[191] = \text{TEMP}[255] = 0$ )  
THEN ZF  $\leftarrow 1$ ;  
ELSE ZF  $\leftarrow 0$ ;

$\text{TEMP}[255:0] \leftarrow \text{SRC}[255:0] \text{ AND NOT } \text{DEST}[255:0]$   
IF ( $\text{TEMP}[63] = \text{TEMP}[127] = \text{TEMP}[191] = \text{TEMP}[255] = 0$ )  
THEN CF  $\leftarrow 1$ ;  
ELSE CF  $\leftarrow 0$ ;  
DEST (unmodified)  
AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow 0$ ;

## 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 Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.
	If VEX.W = 1 for VTESTPS or VTESTPD.

**VZEROALL—Zero All YMM Registers**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.0F.WIG 77	NP	V/V	AVX	Zero all YMM registers.
VZEROALL				

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

**Description**

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation****VZEROALL (VEX.256 encoded version)**

IF (64-bit mode)

```
YMM0[VLMAX-1:0] ← 0
YMM1[VLMAX-1:0] ← 0
YMM2[VLMAX-1:0] ← 0
YMM3[VLMAX-1:0] ← 0
YMM4[VLMAX-1:0] ← 0
YMM5[VLMAX-1:0] ← 0
YMM6[VLMAX-1:0] ← 0
YMM7[VLMAX-1:0] ← 0
YMM8[VLMAX-1:0] ← 0
YMM9[VLMAX-1:0] ← 0
YMM10[VLMAX-1:0] ← 0
YMM11[VLMAX-1:0] ← 0
YMM12[VLMAX-1:0] ← 0
YMM13[VLMAX-1:0] ← 0
YMM14[VLMAX-1:0] ← 0
YMM15[VLMAX-1:0] ← 0
```

ELSE

```
YMM0[VLMAX-1:0] ← 0
YMM1[VLMAX-1:0] ← 0
YMM2[VLMAX-1:0] ← 0
YMM3[VLMAX-1:0] ← 0
YMM4[VLMAX-1:0] ← 0
YMM5[VLMAX-1:0] ← 0
YMM6[VLMAX-1:0] ← 0
YMM7[VLMAX-1:0] ← 0
YMM8-15: Unmodified
```

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

VZEROALL:       \_mm256\_zeroall()

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 8.

**VZEROUPPER—Zero Upper Bits of YMM Registers**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F.WIG 77 VZEROUPPER	NP	V/V	AVX	Zero upper 128 bits of all YMM registers.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

**Description**

The instruction zeros the bits in position 128 and higher of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation****VZEROUPPER**

IF (64-bit mode)

```

YMM0[VLMAX-1:128] ← 0
YMM1[VLMAX-1:128] ← 0
YMM2[VLMAX-1:128] ← 0
YMM3[VLMAX-1:128] ← 0
YMM4[VLMAX-1:128] ← 0
YMM5[VLMAX-1:128] ← 0
YMM6[VLMAX-1:128] ← 0
YMM7[VLMAX-1:128] ← 0
YMM8[VLMAX-1:128] ← 0
YMM9[VLMAX-1:128] ← 0
YMM10[VLMAX-1:128] ← 0
YMM11[VLMAX-1:128] ← 0
YMM12[VLMAX-1:128] ← 0
YMM13[VLMAX-1:128] ← 0
YMM14[VLMAX-1:128] ← 0
YMM15[VLMAX-1:128] ← 0

```

ELSE

```

YMM0[VLMAX-1:128] ← 0
YMM1[VLMAX-1:128] ← 0
YMM2[VLMAX-1:128] ← 0
YMM3[VLMAX-1:128] ← 0
YMM4[VLMAX-1:128] ← 0
YMM5[VLMAX-1:128] ← 0
YMM6[VLMAX-1:128] ← 0
YMM7[VLMAX-1:128] ← 0
YMM8-15: unmodified

```

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

VZEROUPPER:     `_mm256_zeroupper()`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 8.

**WAIT/FWAIT—Wait**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9B	WAIT	NP	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	NP	Valid	Valid	Check pending unmasked floating-point exceptions.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

**Description**

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

CheckForPendingUnmaskedFloatingPointExceptions;

**FPU Flags Affected**

The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM	If CRO.MP[bit 1] = 1 and CRO.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
OF 09	WBINVD	NP	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Writes back all modified cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

### Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
```

### Flags Affected

None.

### Protected Mode Exceptions

- |        |  |
|--------|--|
| #GP(0) | If the current privilege level is not 0. |
| #UD    | If the LOCK prefix is used.              |

### Real-Address Mode Exceptions

- |     |                             |
|-----|-----------------------------|
| #UD | If the LOCK prefix is used. |
|-----|-----------------------------|

**Virtual-8086 Mode Exceptions**

#GP(0) WBINVD cannot be executed at the virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## WRFSBASE/WRGSBASE—Write FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Fea- ture Flag	Description
F3 OF AE /2 WRFSBASE r32	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
REX.W + F3 OF AE /2 WRFSBASE r64	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 OF AE /3 WRGSBASE r32	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
REX.W + F3 OF AE /3 WRGSBASE r64	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

### Operation

FS/GS segment base address ← SRC;

### Flags Affected

None

### C/C++ Compiler Intrinsic Equivalent

```
WRFSBASE: void _writefsbase_u32( unsigned int );
WRFSBASE: _writefsbase_u64( unsigned __int64 );
WRGSBASE: void _writegsbase_u32( unsigned int );
WRGSBASE: _writegsbase_u64( unsigned __int64 );
```

### Protected Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

### Real-Address Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.

#### 64-Bit Mode Exceptions

- #UD                  If the LOCK prefix is used.  
                If CR4.FSGSBASE[bit 16] = 0.  
                If CPUID.07H.0H: EBX.FSGSBASE[bit 0] = 0
- #GP(0)              If the source register contains a non-canonical address.

## WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 30	WRMSR	NP	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 35, “Model-Specific Registers (MSRs)”, in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). Note that WRMSR to the IA32\_TSC\_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

### Operation

$\text{MSR[ECX]} \leftarrow \text{EDX:EAX};$

### Flags Affected

None.

**Protected Mode Exceptions**

- #GP(0)
  - If the current privilege level is not 0.
  - If the value in ECX specifies a reserved or unimplemented MSR address.
  - If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
  - If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32\_DS\_AREA, IA32\_FS\_BASE, IA32\_GS\_BASE, IA32\_KERNEL\_GS\_BASE, IA32\_LSTAR, IA32\_SYSENTER\_EIP, IA32\_SYSENTER\_ESP.
- #UD
  - If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP
  - If the value in ECX specifies a reserved or unimplemented MSR address.
  - If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
  - If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32\_DS\_AREA, IA32\_FS\_BASE, IA32\_GS\_BASE, IA32\_KERNEL\_GS\_BASE, IA32\_LSTAR, IA32\_SYSENTER\_EIP, IA32\_SYSENTER\_ESP.
- #UD
  - If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- #GP(0)
  - The WRMSR instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## XACQUIRE/XRELEASE – Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE <sup>1</sup>	A hint used with an "XACQUIRE-enabled" instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an "XRELEASE-enabled" instruction to end lock elision on the instruction memory operand address.

### NOTES:

1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

### Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The "MOV mem, reg" (Opcode 88H/89H) and "MOV mem, imm" (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Section 15.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

### Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REPNEZ 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 prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,

- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

## Operation

### XACQUIRE

```

IF XACQUIRE-enabled instruction
  THEN
    IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN
      HLE_NEST_COUNT++
      IF (HLE_NEST_COUNT = 1) THEN
        HLE_ACTIVE ← 1
        IF 64-bit mode
          THEN
            restartRIP ← instruction pointer of the XACQUIRE-enabled instruction
          ELSE
            restartEIP ← instruction pointer of the XACQUIRE-enabled instruction
          FI;
        Enter HLE Execution (* record register state, start tracking memory state *)
      FI; (* HLE_NEST_COUNT = 1 *)
      IF ElisionBufferAvailable
        THEN
          Allocate elision buffer
          Record address and data for forwarding and commit checking
          Perform elision
        ELSE
          Perform lock acquire operation transactionally but without elision
        FI;
      ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)
        GOTO HLE_ABORT_PROCESSING
      FI;
    ELSE
      Treat instruction as non-XACQUIRE F2H prefixed legacy instruction
    FI;
  
```

### XRELEASE

```

IF XRELEASE-enabled instruction
  THEN
    IF (HLE_NEST_COUNT > 0)
      THEN
        HLE_NEST_COUNT--
        IF lock address matches in elision buffer THEN
          IF lock satisfies address and value requirements THEN
            Deallocate elision buffer
          ELSE
            GOTO HLE_ABORT_PROCESSING
          FI;
        FI;
        IF (HLE_NEST_COUNT = 0)
          THEN
            IF NoAllocatedElisionBuffer
              THEN

```

```

Try to commit transactional execution
IF fail to commit transactional execution
    THEN
        GOTO HLE_ABORT_PROCESSING;
    ELSE (* commit success *)
        HLE_ACTIVE ← 0
    FI;
ELSE
    GOTO HLE_ABORT_PROCESSING
FI;
FI;
FI; (* HLE_NEST_COUNT > 0 *)
ELSE
    Treat instruction as non-XRELEASE F3H prefixed legacy instruction
FI;

(* For any HLE abort condition encountered during HLE execution *)
HLE_ABORT_PROCESSING:
    HLE_ACTIVE ← 0
    HLE_NEST_COUNT ← 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP ← restartRIP
        ELSE
            EIP ← restartEIP
        FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END

```

## SIMD Floating-Point Exceptions

None

## Other Exceptions

#GP(0) If the use of prefix causes instruction length to exceed 15 bytes.

**XABORT — Transactional Abort**

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature	Description
Flag				
C6 F8 ib	A	V/V	RTM	Causes an RTM abort if in RTM execution
XABORT imm8				

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	NA	NA	NA

**Description**

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

**Operation**

```
XABORT
IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
FI;
```

(\* For any RTM abort condition encountered during RTM execution \*)

```
RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT ← 0;
    RTM_ACTIVE ← 0;
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP;
        ELSE
            EIP ← fallbackEIP;
    FI;
END
```

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

XABORT: void \_xabort(unsigned int);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

#UD            CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0.  
              If LOCK prefix is used.

## XADD—Exchange and Add

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF C0 /r	XADD r/m8, r8	MR	Valid	Valid	Exchange r8 and r/m8; load sum into r/m8.
REX + OF C0 /r	XADD r/m8*, r8*	MR	Valid	N.E.	Exchange r8 and r/m8; load sum into r/m8.
OF C1 /r	XADD r/m16, r16	MR	Valid	Valid	Exchange r16 and r/m16; load sum into r/m16.
OF C1 /r	XADD r/m32, r32	MR	Valid	Valid	Exchange r32 and r/m32; load sum into r/m32.
REX.W + OF C1 /r	XADD r/m64, r64	MR	Valid	N.E.	Exchange r64 and r/m64; load sum into r/m64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (W)	NA	NA

### Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

### IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

### Operation

```
TEMP ← SRC + DEST;
SRC ← DEST;
DEST ← TEMP;
```

### Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Real-Address Mode Exceptions

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

### Virtual-8086 Mode Exceptions

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

## XBEGIN — Transactional Begin

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature	Description
			Flag	
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	NA	NA	NA

### Description

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

### Operation

#### XBEGIN

```

IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
    THEN
        RTM_NEST_COUNT++
        IF RTM_NEST_COUNT = 1 THEN
            IF 64-bit Mode
                THEN
                    fallbackRIP ← RIP + SignExtend64(IMM)
                    (* RIP is instruction following XBEGIN instruction *)
                ELSE
                    fallbackEIP ← EIP + SignExtend32(IMM)
                    (* EIP is instruction following XBEGIN instruction *)
                ENDIF;
            ENDIF;
            IF (64-bit mode)
                THEN IF (fallbackRIP is not canonical)
                    THEN #GP(0)
                ENDIF;
                ELSE IF (fallbackEIP outside code segment limit)
                    THEN #GP(0)
                ENDIF;
            ENDIF;
            RTM_ACTIVE ← 1
            Enter RTM Execution (* record register state, start tracking memory state*)
        ENDIF;
    ENDIF;

```

```

    FI; (* RTM_NEST_COUNT = 1 *)
    ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
        GOTO RTM_ABORT_PROCESSING
    FI;
(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END

```

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

XBEGIN:    unsigned int \_xbegin( void );

### SIMD Floating-Point Exceptions

None

### Protected Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11]=0. If LOCK prefix is used.
#GP(0)	If the fallback address is outside the CS segment.

### Real-Address Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11]=0. If LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11]=0. If LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-bit Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0. If LOCK prefix is used.
-----	--

## INSTRUCTION SET REFERENCE, N-Z

#GP(0) If the fallback address is non-canonical.

## XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
90+rw	XCHG AX, r16	O	Valid	Valid	Exchange r16 with AX.
90+rw	XCHG r16, AX	O	Valid	Valid	Exchange AX with r16.
90+rd	XCHG EAX, r32	O	Valid	Valid	Exchange r32 with EAX.
REX.W + 90+rd	XCHG RAX, r64	O	Valid	N.E.	Exchange r64 with RAX.
90+rd	XCHG r32, EAX	O	Valid	Valid	Exchange EAX with r32.
REX.W + 90+rd	XCHG r64, RAX	O	Valid	N.E.	Exchange RAX with r64.
86 /r	XCHG r/m8, r8	MR	Valid	Valid	Exchange r8 (byte register) with byte from r/m8.
REX + 86 /r	XCHG r/m8*, r8*	MR	Valid	N.E.	Exchange r8 (byte register) with byte from r/m8.
86 /r	XCHG r8, r/m8	RM	Valid	Valid	Exchange byte from r/m8 with r8 (byte register).
REX + 86 /r	XCHG r8*, r/m8*	RM	Valid	N.E.	Exchange byte from r/m8 with r8 (byte register).
87 /r	XCHG r/m16, r16	MR	Valid	Valid	Exchange r16 with word from r/m16.
87 /r	XCHG r16, r/m16	RM	Valid	Valid	Exchange word from r/m16 with r16.
87 /r	XCHG r/m32, r32	MR	Valid	Valid	Exchange r32 with doubleword from r/m32.
REX.W + 87 /r	XCHG r/m64, r64	MR	Valid	N.E.	Exchange r64 with quadword from r/m64.
87 /r	XCHG r32, r/m32	RM	Valid	Valid	Exchange doubleword from r/m32 with r32.
REX.W + 87 /r	XCHG r64, r/m64	RM	Valid	N.E.	Exchange quadword from r/m64 with r64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	AX/EAX/RAX (r, w)	opcode + rd (r, w)	NA	NA
O	opcode + rd (r, w)	AX/EAX/RAX (r, w)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```
TEMP ← DEST;
DEST ← SRC;
SRC ← TEMP;
```

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If either operand is in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## XEND — Transactional End

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
OF 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

### Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

### Operation

#### XEND

```

IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transactional execution
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE ← 0
            END;
        END;
    END;
(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:

```

```

    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
        END;
    END

```

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

XEND:      void \_xend( void );

### SIMD Floating-Point Exceptions

None

### Other Exceptions

- #UD                    CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0.  
                        If LOCK or 66H or F2H or F3H prefix is used.
- #GP(0)                If RTM\_ACTIVE = 0.

## XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 D0	XGETBV	NP	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCRO is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCRO and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, “Managing State Using the XSAVE Feature Set,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Use of any other value for ECX results in a general-protection (#GP) exception.

### Operation

EDX:EAX ← XCR[ECX];

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

XGETBV:      unsigned \_\_int64 \_xgetbv(unsigned int);

### Protected Mode Exceptions

- #GP(0)      If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD      If CPUID.01H:ECX.XSAVE[bit 26] = 0.  
If CR4.OSXSAVE[bit 18] = 0.  
If the LOCK prefix is used.  
If 66H, F3H or F2H prefix is used.

### Real-Address Mode Exceptions

- #GP(0)      If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD      If CPUID.01H:ECX.XSAVE[bit 26] = 0.  
If CR4.OSXSAVE[bit 18] = 0.  
If the LOCK prefix is used.  
If 66H, F3H or F2H prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D7	XLAT <i>m8</i>	NP	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	NP	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	NP	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS: BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operand” form and the “no-operand” form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS: (E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a “short form” of the XLAT instructions. Here also the processor assumes that the DS: (E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF AddressSize = 16
  THEN
    AL ← (DS:BX + ZeroExtend(AL));
  ELSE IF (AddressSize = 32)
    AL ← (DS:EBX + ZeroExtend(AL)); Fl;
  ELSE (AddressSize = 64)
    AL ← (RBX + ZeroExtend(AL));
  Fl;
```

### 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 <i>ib</i>	XOR AL, <i>imm8</i>	I	Valid	Valid	AL XOR <i>imm8</i> .
35 <i>iw</i>	XOR AX, <i>imm16</i>	I	Valid	Valid	AX XOR <i>imm16</i> .
35 <i>id</i>	XOR EAX, <i>imm32</i>	I	Valid	Valid	EAX XOR <i>imm32</i> .
REX.W + 35 <i>id</i>	XOR RAX, <i>imm32</i>	I	Valid	N.E.	RAX XOR <i>imm32</i> ( <i>sign-extended</i> ).
80 /6 <i>ib</i>	XOR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m8</i> XOR <i>imm8</i> .
REX + 80 /6 <i>ib</i>	XOR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m8</i> XOR <i>imm8</i> .
81 /6 <i>iw</i>	XOR <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	<i>r/m16</i> XOR <i>imm16</i> .
81 /6 <i>id</i>	XOR <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	<i>r/m32</i> XOR <i>imm32</i> .
REX.W + 81 /6 <i>id</i>	XOR <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	<i>r/m64</i> XOR <i>imm32</i> ( <i>sign-extended</i> ).
83 /6 <i>ib</i>	XOR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m16</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
83 /6 <i>ib</i>	XOR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m32</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
REX.W + 83 /6 <i>ib</i>	XOR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m64</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
30 / <i>r</i>	XOR <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	<i>r/m8</i> XOR <i>r8</i> .
REX + 30 / <i>r</i>	XOR <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	<i>r/m8</i> XOR <i>r8</i> .
31 / <i>r</i>	XOR <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	<i>r/m16</i> XOR <i>r16</i> .
31 / <i>r</i>	XOR <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	<i>r/m32</i> XOR <i>r32</i> .
REX.W + 31 / <i>r</i>	XOR <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	<i>r/m64</i> XOR <i>r64</i> .
32 / <i>r</i>	XOR <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	<i>r8</i> XOR <i>r/m8</i> .
REX + 32 / <i>r</i>	XOR <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	<i>r8</i> XOR <i>r/m8</i> .
33 / <i>r</i>	XOR <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	<i>r16</i> XOR <i>r/m16</i> .
33 / <i>r</i>	XOR <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	<i>r32</i> XOR <i>r/m32</i> .
REX.W + 33 / <i>r</i>	XOR <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	<i>r64</i> XOR <i>r/m64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AE/XAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

$\text{DEST} \leftarrow \text{DEST} \text{ XOR } \text{SRC};$

## Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

## Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57 /r XORPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Bitwise exclusive-OR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 57 /r VXORPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 57 /r VXORPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical exclusive-OR of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### XORPD (128-bit Legacy SSE version)

DEST[63:0]  $\leftarrow$  DEST[63:0] BITWISE XOR SRC[63:0]  
 DEST[127:64]  $\leftarrow$  DEST[127:64] BITWISE XOR SRC[127:64]  
 DEST[VLMAX-1:128] (Unmodified)

#### VXORPD (VEX.128 encoded version)

DEST[63:0]  $\leftarrow$  SRC1[63:0] BITWISE XOR SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64] BITWISE XOR SRC2[127:64]  
 DEST[VLMAX-1:128]  $\leftarrow$  0

#### VXORPD (VEX.256 encoded version)

DEST[63:0]  $\leftarrow$  SRC1[63:0] BITWISE XOR SRC2[63:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64] BITWISE XOR SRC2[127:64]  
 DEST[191:128]  $\leftarrow$  SRC1[191:128] BITWISE XOR SRC2[191:128]  
 DEST[255:192]  $\leftarrow$  SRC1[255:192] BITWISE XOR SRC2[255:192]

**Intel C/C++ Compiler Intrinsic Equivalent**

XORPD:           `__m128d _mm_xor_pd(__m128d a, __m128d b)`  
VXORPD:          `__m256d _mm256_xor_pd (__m256d a, __m256d b);`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4.

## XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 57 /r XORPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Bitwise exclusive-OR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 57 /r VXORPS <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 57 /r VXORPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical exclusive-OR of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### XORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

#### VXORPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

**VXORPS (VEX.256 encoded version)**

DEST[31:0]  $\leftarrow$  SRC1[31:0] BITWISE XOR SRC2[31:0]  
 DEST[63:32]  $\leftarrow$  SRC1[63:32] BITWISE XOR SRC2[63:32]  
 DEST[95:64]  $\leftarrow$  SRC1[95:64] BITWISE XOR SRC2[95:64]  
 DEST[127:96]  $\leftarrow$  SRC1[127:96] BITWISE XOR SRC2[127:96]  
 DEST[159:128]  $\leftarrow$  SRC1[159:128] BITWISE XOR SRC2[159:128]  
 DEST[191:160]  $\leftarrow$  SRC1[191:160] BITWISE XOR SRC2[191:160]  
 DEST[223:192]  $\leftarrow$  SRC1[223:192] BITWISE XOR SRC2[223:192]  
 DEST[255:224]  $\leftarrow$  SRC1[255:224] BITWISE XOR SRC2[255:224].

**Intel C/C++ Compiler Intrinsic Equivalent**

XORPS:           `_m128 _mm_xor_ps(_m128 a, _m128 b)`  
 VXORPS:          `_m256 _mm256_xor_ps (_m256 a, _m256 b);`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4.

## XRSTOR—Restore Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /5	XRSTOR <i>mem</i>	M	Valid	Valid	Restore state components specified by EDX:EAX from <i>mem</i> .
REX.W+ OF AE /5	XRSTOR64 <i>mem</i>	M	Valid	N.E.	Restore state components specified by EDX:EAX from <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.7, “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 XCMP\_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*.<sup>1</sup>
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE\_BV field in the XSAVE header, XRSTOR initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE\_BV[*i*] = 1, XRSTOR loads state component *i* from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR\_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE\_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

```
RFBM ← XCR0 AND EDX:EAX; /* bitwise logical AND */
COMPRESS ← XCMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
IF in VMX non-root operation
    THEN VMXNR ← 1;
```

- There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

```

    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;

IF COMPMASK[63] = 0
THEN
/* Standard form of XRSTOR */
If RFBM[0] = 1
THEN
    IF RSTORMASK[0] = 1
        THEN load x87 state from legacy region of XSAVE area;
        ELSE initialize x87 state;
    FI;
FI;
If RFBM[1] = 1
THEN
    IF RSTORMASK[1] = 1
        THEN load XMM registers from legacy region of XSAVE area;
        ELSE set all XMM registers to 0;
    FI;
FI;
If RFBM[2] = 1
THEN
    IF RSTORMASK[2] = 1
        THEN load AVX state from extended region (standard format) of XSAVE area;
        ELSE initialize AVX state;
    FI;
FI;
If RFBM[1] = 1 or RFBM[2] = 1
    THEN load MXCSR from legacy region of XSAVE area;
FI;
FI;
ELSE
/* Compacted form of XRSTOR */
IF CPUID.(EAX=0DH,ECX=1);EAX.XSAVENC[bit 1] = 0
    THEN /* compacted form not supported */
        #GP(0);
FI;
If RFBM[0] = 1
THEN
    IF RSTORMASK[0] = 1
        THEN load x87 state from legacy region of XSAVE area;
        ELSE initialize x87 state;
    FI;
FI;
If RFBM[1] = 1
THEN
    IF RSTORMASK[1] = 1
        THEN load SSE state from legacy region of XSAVE area;
        ELSE initialize SSE state;
    FI;
FI;
If RFBM[2] = 1
THEN

```

```

IF RSTORMASK[2] = 1
    THEN load AVX state from extended region (compacted format) of XSAVE area;
    ELSE initialize AVX state;
FI;
FI;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPRESS⟩;

```

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

```

XRSTOR: void _xrstor( void * , unsigned __int64);
XRSTOR: void _xrstor64( void * , unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVE[bit 1] = 0. If the standard form is executed and a bit in XCRO 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 XCRO 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 CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an align- ment check exception might be signaled for a 2-byte misalignment, whereas a general protec- tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

### Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH. If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVE[bit 1] = 0.
-----	---

- If the standard form is executed and a bit in XCRO 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 XCRO 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 CRO.TS[bit 3] = 1.
- #UD            If CPUID.01H:ECX.XSAVE[bit 26] = 0.  
                If CR4.OSXSAVE[bit 18] = 0.  
                If any of the LOCK, 66H, F3H or F2H prefixes is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- |                 |  |
|-----------------|--|
| #GP(0)          | If a memory address is in a non-canonical form.<br><br>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.<br><br>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and<br>CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.<br><br>If the standard form is executed and a bit in XCRO is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.<br><br>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.<br><br>If the compacted form is executed and a bit in XCRO is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.<br><br>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.<br><br>If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.<br><br>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 CRO.TS[bit 3] = 1.  |
| #UD             | If CPUID.01H:ECX.XSAVE[bit 26] = 0.<br>If CR4.OSXSAVE[bit 18] = 0.<br>If any of the LOCK, 66H, F3H or F2H prefixes is used.  |
| #AC             | If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).   |

## XRSTORS—Restore Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C7 /3	XRSTORS <i>mem</i>	M	Valid	Valid	Restore state components specified by EDX:EAX from <i>mem</i> .
REX.W+ 0F C7 /3	XRSTORS64 <i>mem</i>	M	Valid	N.E.	Restore state components specified by EDX:EAX from <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCRO with the IA32\_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.11, “Operation of XRSTORS,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, “Extended Region of an XSAVE Area”).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32\_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE\_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE\_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR\_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE\_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

```

RFBM ← (XCRO OR IA32_XSS) AND EDX:EAX;           /* bitwise logical OR and AND */
COMPmask ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;

```

LAXA ← linear address of XSAVE area;

```

If RFBM[0] = 1
  THEN
    IF RSTORMASK[0] = 1
      THEN load x87 state from legacy region of XSAVE area;
      ELSE initialize x87 state;
    FI;
  FI;
If RFBM[1] = 1
  THEN
    IF RSTORMASK[1] = 1
      THEN load SSE state from legacy region of XSAVE area;
      ELSE initialize SSE state;
    FI;
  FI;
If RFBM[2] = 1
  THEN
    IF RSTORMASK[2] = 1
      THEN load AVX state from extended region (compacted format) of XSAVE area;
      ELSE initialize AVX state;
    FI;
  FI;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPmask⟩;

```

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

```

XRSTORS: void _xrstors( void * , unsigned __int64);
XRSTORS64: void _xrstors64( void * , unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	If CPL > 0. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If bit 63 of the XCOMP_BV field of the XSAVE header is 0. If a bit in XCRO 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 CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check

exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

### Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH. If bit 63 of the XCOMP_BV field of the XSAVE header is 0. If a bit in XCRO 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 CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	If CPL > 0. If a memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If bit 63 of the XCOMP_BV field of the XSAVE header is 0. If a bit in XCRO 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 CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protec-

tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XSAVE—Save Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /4	XSAVE <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> .
REX.W+ OF AE /4	XSAVE64 <i>mem</i>	M	Valid	NE.	Save state components specified by EDX:EAX to <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCRO.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.6, “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.<sup>1</sup>
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVE reads the XSTATE\_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVE writes XSTATE\_BV[*i*] with the value of XINUSE[*i*]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.5.4, “Processor Tracking of XSAVE-Managed State.”) If RFBM[*i*] = 0, XSAVE writes XSTATE\_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE\_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

```
RFBM ← XCRO AND EDX:EAX; /* bitwise logical AND */
OLD_BV ← XSTATE_BV field from XSAVE header;
```

```
IF RFBM[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;
IF RFBM[1] = 1
    THEN store XMM registers into legacy region of XSAVE area;
FI;
```

1. An exception is made for MXCSR and MXCSR\_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.

```

IF RFBM[2] = 1
    THEN store AVX state into extended region of XSAVE area;
FI;
IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

```

XSTATE\_BV field in XSAVE header  $\leftarrow$  (OLD\_BV AND ~RFBM) OR (XINUSE AND RFBM);

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

XSAVE:	<code>void _xsave( void * , unsigned __int64);</code>
XSAVE:	<code>void _xsave64( void * , unsigned __int64);</code>

### 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 CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

### Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XSAVEC—Save Processor Extended States with Compaction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF C7 /4	XSAVEC <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> with compaction.
REX.W+ OF C7 /4	XSAVEC64 <i>mem</i>	M	Valid	N.E.	Save state components specified by EDX:EAX to <i>mem</i> with compaction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCRO.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.9, “Operation of 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[*j*] = 1 and XINUSE[*j*] = 1.<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.5.4, “Processor Tracking of XSAVE-Managed State.”)
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE\_BV field of the XSAVE header.<sup>2,3</sup> (See Section 13.4.2, “XSAVE Header.”) XSAVEC sets bit 63 of the XCOMP\_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE\_BV and XCOMP\_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

```
RFBM ← XCRO AND EDX:EAX; /* bitwise logical AND */
COMPmask ← RFBM OR 80000000_00000000H;
```

IF RFBM[0] = 1 and XINUSE[0] = 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.
  - Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE\_BV field that correspond to bits that are clear in RFBM.
  - There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE\_BV[1] to 1 as long as RFBM[1] = 1.

```

THEN store x87 state into legacy region of XSAVE area;
Fl;
IF RFBM[1] = 1 and (XINUSE[1] = 1 or MXCSR ≠ 1F80H)
    THEN store SSE state into legacy region of XSAVE area;
Fl;
IF RFBM[2] = 1 AND XINUSE[2] = 1
    THEN store AVX state into extended region of XSAVE area;
Fl;

XSTATE_BV field in XSAVE header ← XINUSE AND RFBM;
XCOMP_BV field in XSAVE header ← COMPmask;

```

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVEC:     void _xsavec( void * , unsigned __int64);
XSAVEC64:   void _xsavec64( void * , unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

### Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.

### 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 CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF AE /6 XSAVEOPT <i>mem</i>	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to <i>mem</i> , optimizing if possible.
REX.W + OF AE /6 XSAVEOPT64 <i>mem</i>	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to <i>mem</i> , optimizing if possible.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCRO.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.8, “Operation of XSAVEOPT,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it uses compaction and that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if RFBM[*j*] = 1 and XINUSE[*j*] = 1.<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.5.4, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRTOR or XRSTORS as determined by the internal value XRSTOR\_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEOPT reads the XSTATE\_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If RFBM[*j*] = 1, XSAVEOPT writes XSTATE\_BV[*j*] with the value of XINUSE[*j*]. If RFBM[*j*] = 0, XSAVEOPT writes XSTATE\_BV[*j*] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE\_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

```
RFBM ← XCRO AND EDX:EAX; /* bitwise logical AND */
OLD_BV ← XSTATE_BV field from XSAVE header;
```

1. There is an exception made for MXCSR and MXCSR\_MASK, which belong to state component 1 – SSE. XSAVEOPT always saves these to memory if RFBM[1] = 1 or RFBM[2] = 1, regardless of the value of XINUSE.

```

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPRESS ← 00000000_00000000H;
IF XRSTOR_INFO = <CPL,VMXNR,LAXA,COMPRESS>
    THEN MODOPT ← 1;
    ELSE MODOPT ← 0;
FI;

IF RFBM[0] = 1 and XINUSE[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
    /* might avoid saving if x87 state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 and XINUSE[1]
    THEN store XMM registers into legacy region of XSAVE area;
    /* might avoid saving if XMM registers are not modified and MODOPT = 1 */
FI;
IF RFBM[2] = 1 AND XINUSE[2] = 1
    THEN store AVX state into extended region of XSAVE area;
    /* might avoid saving if AVX state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

```

XSTATE\_BV field in XSAVE header ← (OLD\_BV AND ~RFBM) OR (XINUSE AND RFBM);

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVEOPT: void _xsaveopt( void * , unsigned __int64);
XSAVEOPT: void _xsaveopt64( void * , unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used. If 66H, F3H or F2H prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
-----	--

#NM                    If CRO.TS[bit 3] = 1.  
 #UD                    If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.  
                       If CR4.OSXSAVE[bit 18] = 0.  
                       If the LOCK prefix is used.  
                       If 66H, F3H or F2H prefix is used.

### **Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

### **Compatibility Mode Exceptions**

Same exceptions as in protected mode.

### **64-Bit Mode Exceptions**

#SS(0)                If a memory address referencing the SS segment is in a non-canonical form.  
 #GP(0)                If the memory address is in a non-canonical form.  
                       If a memory operand is not aligned on a 64-byte boundary, regardless of segment.  
 #PF(fault-code)      If a page fault occurs.  
 #NM                    If CRO.TS[bit 3] = 1.  
 #UD                    If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.  
                       If CR4.OSXSAVE[bit 18] = 0.  
                       If the LOCK prefix is used.  
                       If 66H, F3H or F2H prefix is used.

## XSAVES—Save Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF C7 /5	XSAVES <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> with compaction, optimizing if possible.
REX.W+ OF C7 /5	XSAVES64 <i>mem</i>	M	Valid	N.E.	Save state components specified by EDX:EAX to <i>mem</i> with compaction, optimizing if possible.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCRO with the IA32\_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.10, “Operation of XSAVES,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32\_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.5.4, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRTOR or XRSTORS as determined by XRSTOR\_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE\_BV field of the XSAVE header.<sup>2</sup> (See Section 13.4.2, “XSAVE Header.”) XSAVES sets bit 63 of the XCOMP\_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE\_BV and XCOMP\_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

- 
- 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.
  - There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE\_BV[1] to 1 as long as RFBM[1] = 1.

## Operation

```

RFBM ← XCRO AND EDX:EAX; /* bitwise logical AND */
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← RFBM OR 80000000_00000000H;
IF XRSTOR_INFO = {CPL,VMXNR,LAXA,COMPMASK}
    THEN MODOPT ← 1;
    ELSE MODOPT ← 0;
FI;

IF RFBM[0] = 1 and XINUSE[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
    /* might avoid saving if x87 state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 and (XINUSE[1] = 1 or MXCSR ≠ 1F80H)
    THEN store SSE state into legacy region of XSAVE area;
    /* might avoid saving if SSE state is not modified and MODOPT = 1 */
FI;
IF RFBM[2] = 1 AND XINUSE[2] = 1
    THEN store AVX state into extended region of XSAVE area;
    /* might avoid saving if AVX state is not modified and MODOPT = 1 */
FI;

XSTATE_BV field in XSAVE header ← XINUSE AND RFBM;
XCOMP_BV field in XSAVE header ← COMPMASK;

```

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVES:   void _xsaves( void * , unsigned __int64);
XSAVES64: void _xsaves64( void * , unsigned __int64);

```

## Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an align-

ment 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 CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CRO.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XSETBV—Set Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 D1	XSETBV	NP	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCRO is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCRO (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 XCRO[2] (AVX state) while clearing XCRO[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, "Enabling the XSAVE Feature Set and XSAVE-Supported Features," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

### Operation

$\text{XCR}[\text{ECX}] \leftarrow \text{EDX:EAX};$

### 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 XCRO.
  - If an attempt is made to set XCRO[2:1] to 10b.
- #UD
  - If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.
  - If 66H, F3H or F2H prefix is used.

### Real-Address Mode Exceptions

- #GP      If an invalid XCR is specified in ECX.
  - If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
  - If an attempt is made to clear bit 0 of XCRO.
  - If an attempt is made to set XCRO[2:1] to 10b.
- #UD      If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.
  - If 66H, F3H or F2H prefix is used.

### Virtual-8086 Mode Exceptions

- #GP(0)    The XSETBV instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## XTEST – Test If In Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode	CPUID Feature Support	Flag	Description
OF 01 D6 XTEST	A	V/V	HLE or RTM		Test if executing in a transactional region

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

### Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

### Operation

```

XTEST
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF ← 0
    ELSE
        ZF ← 1
FI;
```

### Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

### Intel C/C++ Compiler Intrinsic Equivalent

XTEST: int \_xtest( void );

### SIMD Floating-Point Exceptions

None

### Other Exceptions

#UD CPUID.(EAX=7, ECX=0):HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0.  
If LOCK or 66H or F2H or F3H prefix is used.

## INSTRUCTION SET REFERENCE, N-Z

### 5.1 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)<sup>1</sup>. The system executive may be based on a Virtual Machine Monitor (VMM), a measured VMM is referred to as MVMM<sup>2</sup>.
- 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.

### 5.2 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, OF 37).

#### 5.2.1 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[Bit 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 5-1.

---

1. See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide*.  
2. An MVMM is sometimes referred to as a measured launched environment (MLE). See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide*

**Table 5-1. 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
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]
63:16	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 5, "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.

## 5.2.2 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.

### 5.2.2.1 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 5-2 lists the currently available GETSEC leaf functions.

**Table 5-2. GETSEC Leaf Functions**

<b>Index (EAX)</b>	<b>Leaf function</b>	<b>Description</b>
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

### 5.2.2.2 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.

### 5.2.2.3 GETSEC[EXITAC]

GETSEC[EXITAC] takes the processor out of . 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.

### 5.2.2.4 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].

### **5.2.2.5 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.

### **5.2.2.6 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.

### **5.2.2.7 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.

### **5.2.2.8 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.

## **5.2.3 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. *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 wakeup 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.

## 5.3 GETSEC LEAF FUNCTIONS

This section provides detailed descriptions of each leaf function of the GETSEC instruction. GETSEC is available only if CPUID.01H:ECX[Bit 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<sup>1</sup> passed as input parameters to GETSEC functions and to register results returned as output parameters.

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<sup>2</sup> 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

- 
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
  2. 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.

## SAFER MODE EXTENSIONS REFERENCE

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.

## GETSEC[CAPABILITIES] - Report the SMX Capabilities

Opcode	Instruction	Description
OF 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 5-3.

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 5-3. 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
SENDER	4	GETSEC[SENDER] 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 (SENDER Available)
    THEN BitVector[SENDER]← 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/REP NZ and REP/REPE/REPZ)
Operand size	Causes #UD
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
OF 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 insure 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 5-4. 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 5-4. Register State Initialization after GETSEC[ENTERACCS]**

Register State	Initialization Status	Comment
CRO	PG←0, AM←0, WP←0: Others unchanged	Paging, Alignment Check, Write-protection are disabled
CR4	MCE←0: Others unchanged	Machine Check Exceptions 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 5-5 for example	The number of initialized fields may change due to processor implementation

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.

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 5-5.). 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 5-5. IA32\_MISC\_ENABLE MSR Initialization<sup>1</sup> 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. <sup>2</sup>
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
        (CRO.PE=0) or (CRO.CD=1) or (CRO.NW=1) or (CRO.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) CRO.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*)

```

## SAFER MODE EXTENSIONS REFERENCE

```

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 ACEEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEEntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEEntryPoint >= ACSIZE) OR (ACEEntryPoint < (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);
CRO.[PG.AM.WP]← 0;
CR4.MCE← 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];
CSSEL← ACRAM[SegSel];
CS.BASE← 0;
CS.LIMIT← FFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DSSEL← ACRAM[SegSel]+8;
DS.BASE← 0;
DS.LIMIT← FFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP← ACEEntryPoint;
END;

```

## Flags Affected

All flags are cleared.

## Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNEZ and REP/REPE/REPZ)
Operand size	Causes #UD
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 CRO.CD = 1 or CRO.NW = 1 or CRO.NE = 0 or CRO.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 CRO.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
OF 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) )
        (CRO.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);
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;
EIP← tempEIP;
END;
```

## Flags Affected

None.

## Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNEZ and REP/REPE/REPZ)
Operand size	Causes #UD

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
OF 37 (EAX=4)	GETSEC[SENTER]	<p>Launch a measured environment</p> <p>EBX holds the SINIT authenticated code module physical base address.</p> <p>ECX holds the SINIT authenticated code module size (bytes).</p> <p>EDX controls the level of functionality supported by the measured environment launch.</p>

### 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 CRO.CD and NW bits.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CRO.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 insure consistent handling of SIPI messages, the processor executing the GETSEC[SENTER] instruction must also be designated the BSP (boot-strap processor) as defined by A32\_APIC\_BASE.BSP (Bit 8).
- EDX must be initialized to a setting supportable by the processor. Unless enumeration by the GETSEC[PATTERNS] 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/accessible for DS, SS, and ES, while execute/read/accessible 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 5-6 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 5-6. Register State Initialization after GETSEC[SENTER] and GETSEC[WAKEUP]**

Register State	ILP after GETSEC[SENTER]	RLP after GETSEC[WAKEUP]
CRO	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 <sup>1</sup> ]	[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 5-5	See Table 5-5
IA32_SMM_MONITOR_CTL	Bit 2←0	Bit 2←0

**NOTES:**

1. See *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 5-6. 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 5-5 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 5-1.

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
          (CRO.PE=0) or (CRO.CD=1) or (CRO.NW=1) or (CRO.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_CR_FEATURE_CONTROL[0]=0) or (IA32_CR_FEATURE_CONTROL[15]=0) or
          ((IA32_CR_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;

```

```

Fl;
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 SignalSENDER message);

```

**TXT\_SETER\_MSG\_EVENT (ILP & RLP):**

```

Mask and clear SignalSENDER 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);
    Fl;
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);
    Fl;

```

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(SETERAck);
IF (logical processor is not ILP)
    THEN GOTO RLP_SETER_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
    DONE← TXT.READ(LT.STS);
    WHILE (not DONE);
    SignalTXTMsg(SETERContinue);
    SignalTXTMsg(ProcessorHold);
    FOR I=ACBASE to ACBASE+ACSIZE-1 DO
        ACRAM[I-ACBASE].ADDR← I;
        ACRAM[I-ACBASE].DATA← LOAD(I);
    OD;

```

## SAFER MODE EXTENSIONS REFERENCE

```

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(#UnexpectedHIT);
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 ACEEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEEntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEEntryPoint >= ACSIZE) or (ACEEntryPoint < (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 OFED44008H, TMP_DATA_BUFFER_4 is located at OFED44080H *)
        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;
CRO.[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\_SETER\_ROUTINE: (RLP only)**

Mask SMI, INIT, A20M, and NMI external pin events  
 Unmask SignalWAKEUP event;  
 Wait for SignalSETERContinue 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
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 CRO.CD = 1 or CRO.NW = 1 or CRO.NE = 0 or CRO.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
OF 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 (CRO.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 inter-operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or in VMX operation.
- To insure 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
        (CRO.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/REP NZ and REP/REPE/REPZ)
Operand size	Causes #UD
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 CRO.PE = 0 or CPL > 0 or EFLAGS.VM = 1. If in VMX root operation. If the initiating processor is not designated as the 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
OF 37 (EAX=6)	GETSEC[PARAMETERS]	<i>Report the SMX Parameters</i> <i>The parameters index is input in EBX with the result returned in EAX, EBX, and ECX.</i>

### 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 5-7 defines the parameter types supported in current and future implementations.

**Table 5-7. 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 5-8)	Reserved	Reserved
6-31	Undefined	Reserved (unmodified)	Reserved (unmodified)	Reserved (unmodified)

**Table 5-8. TXT Feature Extensions Flags**

<b>Bit</b>	<b>Definition</b>	<b>Description</b>
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 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 5-9.

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 capa-

bility 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 5-9. 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)
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 5-10.

**Table 5-10. 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 5-7 and Table 5-8*)
ELSE (* unsupported index*)
    EAX" 00000000h;
END;

```

## Flags Affected

None.

## Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNEZ and REP/REPE/REPZ)
Operand size	Causes #UD
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
OF 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 5-11). 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 5-11 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 5-11. 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 ((CRO.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/REPNEZ and REP/REPE/REPZ)
Operand size	Causes #UD
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 CRO.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
OF 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 5-12.

**Table 5-12. 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 5-6). Certain states from the content of Table 5-12 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 5-12).

**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 ((CRO.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);
CRO.[PG,CD,NW,AM,WP] ← 0;
CRO.[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/REP NZ and REP/REPE/REPZ)
Operand size	Causes #UD
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 CRO.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.

## 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.

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)).

## OPCODE MAP

- 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.vvv 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.

## OPCODE MAP

- 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 0FA405000000003H 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 OF38H or OF3A. 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 (OF38H or OF3AH). 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 OF3AH 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 OF, OF38H, OF3AH, 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 OF0B opcode (UD2 instruction) or the OFB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table 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.

This page was  
intentionally left  
blank.

**Table A-2. One-byte Opcode Map: (00H – F7H) \***

	0	1	2	3	4	5	6	7
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES <sup>i64</sup>	POP ES <sup>i64</sup>
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS <sup>i64</sup>	POP SS <sup>i64</sup>
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA <sup>i64</sup>
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA <sup>i64</sup>
4	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA <sup>i64</sup> /PUSHAD <sup>i64</sup>	POPA <sup>i64</sup> /POPAD <sup>i64</sup>	BOUND <sup>i64</sup> Gv, Ma	ARPL <sup>i64</sup> Ew, Gw MOVSD <sup>i64</sup> Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1 <sup>1A</sup>				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib <sup>i64</sup>	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, 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	CMP/S/B Xb, Yb	CMP/S/W/D Xv, Yv	
B	AL/R8L, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2 <sup>1A</sup>		near RET <sup>f64</sup> Iw	near RET <sup>f64</sup>	LES <sup>i64</sup> Gz, Mp VEX+2byte	LDS <sup>i64</sup> Gz, Mp VEX+1byte	Grp 11 <sup>1A</sup> - MOV	
D	Shift Grp 2 <sup>1A</sup>		Eb, CL	Ev, CL	AAM <sup>i64</sup> Ib	AAD <sup>i64</sup> Ib		XLAT/ XLATB
E	LOOPNE <sup>i64</sup> /LOOPNZ <sup>i64</sup> Jb	LOOP <sup>i64</sup> /LOOPZ <sup>i64</sup> Jb	LOOP <sup>i64</sup> Jb	JrCXZ <sup>f64</sup> /Jb	IN AL, Ib	eAX, Ib	lb, AL	lb, eAX
F	LOCK (Prefix)		REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3 <sup>1A</sup>	
						Eb	Ev	

Table A-2. One-byte Opcode Map: (08H – FFH) \*

	8	9	A	B	C	D	E	F
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH CS <sup>64</sup>	2-byte escape (Table A-3)
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH DS <sup>64</sup>	POP DS <sup>64</sup>
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=CS (Prefix)	DAS <sup>64</sup>
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=DS (Prefix)	AAS <sup>64</sup>
4	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	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH <sup>d64</sup> lz	IMUL Gv, Ev, Iz	PUSH <sup>d64</sup> lb	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
8	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A <sup>1A</sup> POP <sup>d64</sup> Ev
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	far CALL <sup>f64</sup> Ap	FWAIT/ WAIT	PUSHF/D/Q <sup>d64</sup> / Fv	POPF/D/Q <sup>d64</sup> / Fv	SAHF	LAHF
A	TEST AL, Ib	rAX, Iz	STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Yv
B	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 <sup>d64</sup>	far RET Iw	far RET	INT 3	INT Ib	INTO <sup>64</sup>	IRET/D/Q
D								
E	near CALL <sup>f64</sup> Jz	near <sup>f64</sup> Jz	JMP far <sup>f64</sup> Ap	short <sup>f64</sup> Jb	IN AL, DX	eAX, DX	OUT DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 <sup>1A</sup>	INC/DEC Grp 5 <sup>1A</sup>

**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 <sup>1A</sup>	Grp 7 <sup>1A</sup>	LAR Gv, Ew	LSL Gv, Ew		SYSCALL <sup>064</sup>	CLTS	SYSRET <sup>064</sup>	
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 <sup>v1</sup> Vdq, Hq, Mq vmovlhps Vdq, Hq, Uq	vmovhps <sup>v1</sup> Mq, Vq	
	66	vmovupd Vpd, Wpd	vmovupd Wpd, Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx, Hx, Wx	vunpckhpd Vx, Hx, Wx	vmovhpdv <sup>v1</sup> Vdq, Hq, Mq	vmovhpdv <sup>v1</sup> Mq, Vq	
	F3	vmovss Vx, Hx, Wss	vmovss Wss, Hx, Vss	vmovsdup 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		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		vsqrts Vss, Hss, Wss	vrsqrts 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 <sup>1A</sup> )		(Grp 13 <sup>1A</sup> )	(Grp 14 <sup>1A</sup> )	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper <sup>v</sup> vzeroall <sup>v</sup>
	66	vphufd Vx, Wx, Ib					vpcmpeqb Vx, Hx, Wx	vpcmpeqw Vx, Hx, Wx	vpcmpeqd Vx, Hx, Wx	
	F3	vphufhw Vx, Wx, Ib								
	F2	vphuflw 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 <sup>1B</sup>		prefetchw(/1) Ev		
1		Prefetch <sup>1C</sup> (Grp 16 <sup>1A</sup> )							NOP /0 Ev
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvtps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd, Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvtpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtss2ss Vss, Hss, Ey		vcvtss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvtsi2sd Vsd, Hsd, Ey		vcvtsd2si Gy, Wsd	vcvtsd2si Gy, Wsd		
3		3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulpss 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	vcvtps2dq 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		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A		PUSH <sup>d64</sup> FS	POP <sup>d64</sup> FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B		CMPXCHG Eb, Gb	Ev, Gv	LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX Gv, Eb	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 <sup>1A</sup>	
	66		vcmppd Vpd,Hpd,Wpd,Ib		vpinsrw Vdq,Hdq,Ry/Mw,Ib	vpxtrw Gd, Udq, Ib	vshufpd Vpd,Hpd,Wpd,Ib		
	F3		vcmpss Vss,Hss,Wss,Ib						
	F2		vcmpsd Vsd,Hsd,Wsd,Ib						
D		psrlw Pq, Qq	psrid Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq	
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrid Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmoveq 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	vcvtvpd2dq Vx, Wpd	vmoventdq 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	vdddqu 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		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A		PUSH <sup>d64</sup> GS	POP <sup>d64</sup> GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15 <sup>1A</sup> ) <sup>1C</sup>	IMUL Gv, Ev
B	JMPE (reserved for emulator on IPF)	Grp 10 <sup>1A</sup> Invalid Opcode <sup>1B</sup>	Grp 8 <sup>1A</sup> Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb	Gv, Ew	
	F3	POPCNT Gv, Ev		TZCNT Gv, Ev	LZCNT Gv, Ev				
C		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	padn Pq, Qq
	66	vpsubusb Vx, Hx, Wx	vpsubusw Vx, Hx, Wx	vpmminub Vx, Hx, Wx	vpand Vx, Hx, Wx	vpaddusb Vx, Hx, Wx	vpaddusw Vx, Hx, Wx	vpmmaxub Vx, Hx, Wx	vpandn Vx, Hx, Wx
	F3								
	F2								
E		psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsbt Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq
	66	vpsubsb Vx, Hx, Wx	vpsubsw Vx, Hx, Wx	vpmminsw Vx, Hx, Wx	vpnor Vx, Hx, Wx	vpaddsb Vx, Hx, Wx	vpaddsw Vx, Hx, Wx	vpmmaxsw 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	paddt Pq, Qq	
	66	vpsubbb Vx, Hx, Wx	vpsubww Vx, Hx, Wx	vpsubdd Vx, Hx, Wx	vpsubqq Vx, Hx, Wx	vpaddbb Vx, Hx, Wx	vpaddww Vx, Hx, Wx	vpadddt 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 OF 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	vphufb 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			vcvtpbh2ps <sup>v</sup> Vx, Wx, Ib	blendvps Vdq, Wdq	blendvpd Vdq, Wdq	vpermps <sup>v</sup> Vqq, Hqq, Wqq	vptest Vx, Wx
2	66	vpmovsxwb 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 <sup>v</sup> Vqq, Hqq, Wqq	vpcmpgtq Vx, Hx, Wx
4	66	vpmulld Vx, Hx, Wx	vphminposuw Vdq, Wdq				vpsrlvd/q <sup>v</sup> Vx, Hx, Wx	vpsravd <sup>v</sup> Vx, Hx, Wx	vpslvd/q <sup>v</sup> Vx, Hx, Wx
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq	INVP CID Gy, Mdq					
9	66	vgatherdd/q <sup>v</sup> Vx,Hx,Wx	vgatherqd/q <sup>v</sup> Vx,Hx,Wx	vgatherdps/d <sup>v</sup> Vx,Hx,Wx	vgatherqps/d <sup>v</sup> Vx,Hx,Wx			vfmaddsub132ps/d <sup>v</sup> Vx,Hx,Wx	vfmbsubadd132ps/d <sup>v</sup> Vx,Hx,Wx
A	66							vfmaddsub213ps/d <sup>v</sup> Vx,Hx,Wx	vfmbsubadd213ps/d <sup>v</sup> Vx,Hx,Wx
B	66							vfmaddsub231ps/d <sup>v</sup> Vx,Hx,Wx	vfmbsubadd231ps/d <sup>v</sup> Vx,Hx,Wx
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy	ANDN <sup>v</sup> Gy, By, Ey	Grp 17 <sup>1A</sup>		BZHI <sup>v</sup> Gy, Ey, By		BEXTR <sup>v</sup> Gy, Ey, By
	66	MOVBE Gw, Mw	MOVBE Mw, Gw					ADCX Gy, Ey	SHLX <sup>v</sup> Gy, Ey, By
	F3							PEXT <sup>v</sup> Gy, By, Ey	ADOX Gy, Ey
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey					PDEP <sup>v</sup> Gy, By, Ey	MULX <sup>v</sup> By,Gy,rDX,Ey
	66 & F2	CRC32 Gd, Eb	CRC32 Gd, Ew						SHRX <sup>v</sup> Gy, Ey, By

**Table A-4. Three-byte Opcode Map: 08H – FFH (First Two Bytes are OF 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 <sup>V</sup> Vx,Hx,Wx	vpermilpd <sup>V</sup> Vx,Hx,Wx	vtestps <sup>V</sup> Vx, Wx	vtestpd <sup>V</sup> Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss <sup>V</sup> Vx, Wd	vbroadcastsd <sup>V</sup> Vqq, Wq	vbroadcastf128 <sup>V</sup> Vqq, Mdq		vpabsb Vx, Wx	vpabsw Vx, Wx	vpaBSD Vx, Wx	
2	66	vpmuldq Vx, Hx, Wx	vpcmpeqq Vx, Hx, Wx	vmovntdq Vx, Mx	vpackusdw Vx, Hx, Wx	vmaskmovps <sup>V</sup> Vx,Hx,Mx	vmaskmovpd <sup>V</sup> Vx,Hx,Mx	vmaskmovps <sup>V</sup> Mx,Hx,Vx	vmaskmovpd <sup>V</sup> Mx,Hx,Vx
3	66	vpminsb Vx, Hx, Wx	vpminsd Vx, Hx, Wx	vpmuinw Vx, Hx, Wx	vpmuinud Vx, Hx, Wx	vpmaxsb Vx, Hx, Wx	vpmaxsd Vx, Hx, Wx	vpmaxuw Vx, Hx, Wx	vpmaxud Vx, Hx, Wx
4									
5	66	vpbroadcastd <sup>V</sup> Vx, Wx	vpbroadcastq <sup>V</sup> Vx, Wx	vbroadcasti128 <sup>V</sup> Vqq, Mdq					
6									
7	66	vpbroadcastb <sup>V</sup> Vx, Wx	vpbroadcastw <sup>V</sup> Vx, Wx						
8	66					vpmaskmovd/q <sup>V</sup> Vx,Hx,Mx		vpmaskmovd/q <sup>V</sup> Mx,Vx,Hx	
9	66	vfmadd132ps/d <sup>V</sup> Vx, Hx, Wx	vfmadd132ss/d <sup>V</sup> Vx, Hx, Wx	vfmsub132ps/d <sup>V</sup> Vx, Hx, Wx	vfmsub132ss/d <sup>V</sup> Vx, Hx, Wx	vfnmadd132ps/d <sup>V</sup> Vx, Hx, Wx	vfnmadd132ss/d <sup>V</sup> Vx, Hx, Wx	vfnmsub132ss/d <sup>V</sup> Vx, Hx, Wx	
A	66	vfmadd213ps/d <sup>V</sup> Vx, Hx, Wx	vfmadd213ss/d <sup>V</sup> Vx, Hx, Wx	vfmsub213ps/d <sup>V</sup> Vx, Hx, Wx	vfmsub213ss/d <sup>V</sup> Vx, Hx, Wx	vfnmadd213ps/d <sup>V</sup> Vx, Hx, Wx	vfnmadd213ss/d <sup>V</sup> Vx, Hx, Wx	vfnmsub213ss/d <sup>V</sup> Vx, Hx, Wx	
B	66	vfmadd231ps/d <sup>V</sup> Vx, Hx, Wx	vfmadd231ss/d <sup>V</sup> Vx, Hx, Wx	vfmsub231ps/d <sup>V</sup> Vx, Hx, Wx	vfmsub231ss/d <sup>V</sup> Vx, Hx, Wx	vfnmadd231ps/d <sup>V</sup> Vx, Hx, Wx	vfnmadd231ss/d <sup>V</sup> Vx, Hx, Wx	vfnmsub231ps/d <sup>V</sup> Vx, Hx, Wx	vfnmsub231ss/d <sup>V</sup> Vx, Hx, Wx
C									
D	66				VAESIMC Vdq, Hdq, Wdq	VAESENCLAST Vdq, Hdq, Wdq	VAESENCLAST Vdq, Hdq, Wdq	VAESDEC Vdq, Hdq, Wdq	VAESDECLAST Vdq, Hdq, Wdq
E									
F									
	66								
	F3								
	F2								
	66 & F2								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**Table 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 <sup>v</sup> Vqq, Wqq, Ib	vpermpd <sup>v</sup> Vqq, Wqq, Ib	vpblendd <sup>v</sup> Vx, Hx, Wx, Ib		vpermilps <sup>v</sup> Vx, Wx, Ib	vpermilpd <sup>v</sup> Vx, Wx, Ib	vperm2f128 <sup>v</sup> Vqq, Hqq, Wqq, Ib	
1	66					vpextrb Rd/Mb, Vdq, Ib	vpextrw Rd/Mw, Vdq, Ib	vpxtrd/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 <sup>v</sup> 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 <sup>v</sup> Gy, Ey, Ib							

**Table A-5. Three-byte Opcode Map: 08H – FFH (First Two Bytes are OF 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 <sup>v</sup> Vqq,Hqq,Wqq,Ib	vextractf128 <sup>v</sup> Wdq,Vqq,Ib				vcvtps2ph <sup>v</sup> Wx, Vx, Ib		
2									
3	66	vinserti128 <sup>v</sup> Vqq,Hqq,Wqq,Ib	vextracti128 <sup>v</sup> Wdq,Vqq,Ib						
4	66			vblendvps <sup>v</sup> Vx,Hx,Wx,Lx	vblendvpd <sup>v</sup> Vx,Hx,Wx,Lx	vblendvb <sup>v</sup> Vx,Hx,Wx,Lx			
5									
6									
7									
8									
9									
A									
B									
C									
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## 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.

mod	nnn	R/M
-----	-----	-----

**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 tells us 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 lb/lz		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 <sup>f64</sup> Ev	far CALL Ep	near JMP <sup>f64</sup> Ev	far JMP Mp	PUSH <sup>d64</sup> Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011)	XGETBV (000) XSETBV (001) VMFUNC (100) XEND (101) XTEST (110)					SWAPGS ^064(000) RDTSCP (001)
0F BA	8	mem, 11B						BT	BTS	BTR	BTC
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
			F3							VMXON Mq	VMPTRST Mq
			11B							RDRAND Rv	RDSEED Rv
0F B9	10	mem									
		11B									
C6	11	mem		MOV Eb, Ib							
		11B									XABORT (000) Ib
C7	11	mem		MOV Ev, Iz							
		11B									XBEGIN (000) Jz
0F 71	12	mem						psrlw Nq, Ib	psraw Nq, Ib	pslw Nq, Ib	
		11B					66	vpsrlw Hx,Ux,Ib	vpsraw Hx,Ux,Ib	vpsllw Hx,Ux,Ib	
0F 72	13	mem						psrld Nq, Ib	psrad Nq, Ib	pslld Nq, Ib	
		11B					66	vpsrld Hx,Ux,Ib	vpsrad Hx,Ux,Ib	vpslld Hx,Ux,Ib	
0F 73	14	mem						psrlq Nq, Ib		pslq Nq, Ib	
		11B					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	cflush
		11B	F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry			lfence	mfence
											sfence
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11B									
VEX.0F38 F3	17	mem		BLSR <sup>V</sup> By, Ey	BLSMSK <sup>V</sup> By, Ey	BLSI <sup>V</sup> By, Ey					
		11B									

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## 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.

## OPCODE MAP

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	FLDEN 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	FLDP1	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.

## OPCODE MAP

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.

## OPCODE MAP

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.

## OPCODE MAP

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.

This page was  
intentionally left  
blank.

## 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)

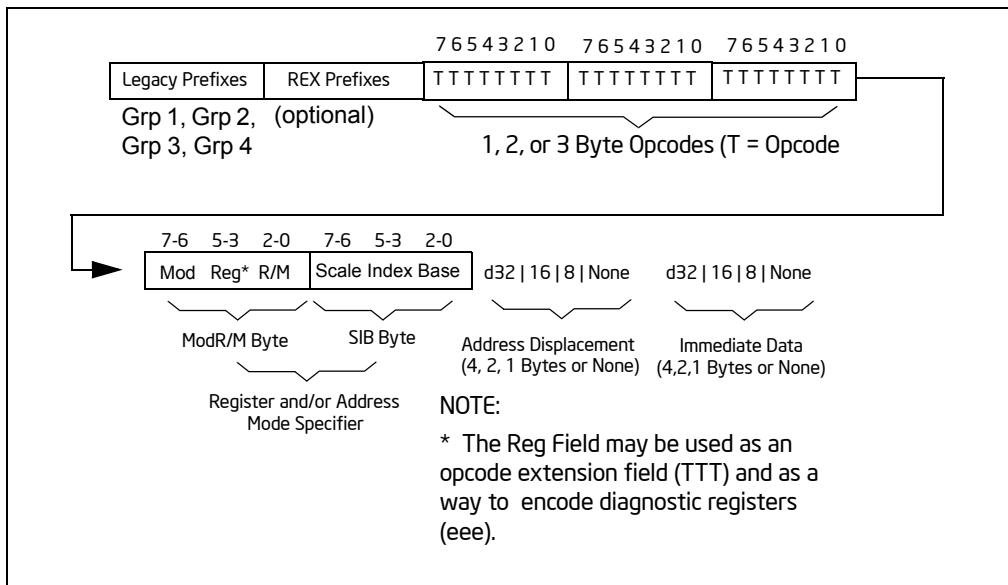


Figure B-1. General Machine Instruction Format

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 new 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			Register Specified by reg Field During 32-Bit Data Operations		
reg	Function of w Field		reg	Function of w Field	
	When w = 0	When w = 1		When w = 0	When w = 1
000	AL	AX	000	AL	EAX
001	CL	CX	001	CL	ECX
010	DL	DX	010	DL	EDX
011	BL	BX	011	BL	EBX
100	AH	SP	100	AH	ESP
101	CH	BP	101	CH	EBP
110	DH	SI	110	DH	ESI
111	BH	DI	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	RPB
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			Register Specified by reg Field During 32-Bit Data Operations		
reg	Function of w Field		reg	Function of w Field	
	When w = 0	When w = 1		When w = 0	When w = 1
000	AL	AX	000	AL	EAX
001	CL	CX	001	CL	ECX
010	DL	DX	010	DL	EDX
011	BL	BX	011	BL	EBX
100	AH <sup>7</sup>	SP	100	AH*	ESP
101	CH <sup>7</sup>	BP	101	CH*	EBP
110	DH <sup>7</sup>	SI	110	DH*	ESI
111	BH <sup>7</sup>	DI	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	3-Bit sreg3 Field	Segment Register Selected
00	ES	000	ES
01	CS	001	CS
10	SS	010	SS
11	DS	011	DS
		100	FS
		101	GS
		110	Reserved <sup>7</sup>
		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 through 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 <sup>7</sup>	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 ttt 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**

<b>tttn</b>	<b>Mnemonic</b>	<b>Condition</b>
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**

<b>d</b>	<b>Source</b>	<b>Destination</b>
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**

<b>Symbol</b>	<b>Note</b>
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
<b>AAA - ASCII Adjust after Addition</b>	0011 0111
<b>AAD - ASCII Adjust AX before Division</b>	1101 0101 : 0000 1010
<b>AAM - ASCII Adjust AX after Multiply</b>	1101 0100 : 0000 1010
<b>AAS - ASCII Adjust AL after Subtraction</b>	0011 1111
<b>ADC - ADD with Carry</b>	
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
<b>ADD - Add</b>	
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
<b>AND - Logical AND</b>	
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
<b>ARPL - Adjust RPL Field of Selector</b>	
from register	0110 0011 : 11 reg1 reg2
from memory	0110 0011 : mod reg r/m
<b>BOUND - Check Array Against Bounds</b>	0110 0010 : mod <sup>A</sup> reg r/m

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>BSF - Bit Scan Forward</b>	
register1, register2	0000 1111 : 1011 1100 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1100 : mod reg r/m
<b>BSR - Bit Scan Reverse</b>	
register1, register2	0000 1111 : 1011 1101 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1101 : mod reg r/m
<b>BSWAP - Byte Swap</b>	0000 1111 : 1100 1 reg
<b>BT - Bit Test</b>	
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
<b>BTC - Bit Test and Complement</b>	
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
<b>BTR - Bit Test and Reset</b>	
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
<b>BTS - Bit Test and Set</b>	
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
<b>CALL - Call Procedure (in same segment)</b>	
direct	1110 1000 : full displacement
register indirect	1111 1111 : 11 010 reg
memory indirect	1111 1111 : mod 010 r/m
<b>CALL - Call Procedure (in other segment)</b>	
direct	1001 1010 : unsigned full offset, selector
indirect	1111 1111 : mod 011 r/m
<b>CBW - Convert Byte to Word</b>	1001 1000
<b>CDQ - Convert Doubleword to Qword</b>	1001 1001
<b>CLC - Clear Carry Flag</b>	1111 1000
<b>CLD - Clear Direction Flag</b>	1111 1100

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>CLI - Clear Interrupt Flag</b>	1111 1010
<b>CLTS - Clear Task-Switched Flag in CR0</b>	0000 1111 : 0000 0110
<b>CMC - Complement Carry Flag</b>	1111 0101
<b>CMP - Compare Two Operands</b>	
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
<b>CMPS/CMPSB/CMPSW/CMPSD - Compare String Operands</b>	1010 011w
<b>CMPXCHG - Compare and Exchange</b>	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
<b>CPUID - CPU Identification</b>	0000 1111 : 1010 0010
<b>CWD - Convert Word to Doubleword</b>	1001 1001
<b>CWDE - Convert Word to Doubleword</b>	1001 1000
<b>DAA - Decimal Adjust AL after Addition</b>	0010 0111
<b>DAS - Decimal Adjust AL after Subtraction</b>	0010 1111
<b>DEC - Decrement by 1</b>	
register	1111 111w : 11 001 reg
register (alternate encoding)	0100 1 reg
memory	1111 111w : mod 001 r/m
<b>DIV - Unsigned Divide</b>	
AL, AX, or EAX by register	1111 011w : 11 110 reg
AL, AX, or EAX by memory	1111 011w : mod 110 r/m
<b>HLT - Halt</b>	1111 0100
<b>IDIV - Signed Divide</b>	
AL, AX, or EAX by register	1111 011w : 11 111 reg
AL, AX, or EAX by memory	1111 011w : mod 111 r/m
<b>IMUL - Signed Multiply</b>	
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

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>IN - Input From Port</b>	
fixed port	1110 010w : port number
variable port	1110 110w
<b>INC - Increment by 1</b>	
reg	1111 111w : 11 000 reg
reg (alternate encoding)	0100 0 reg
memory	1111 111w : mod 000 r/m
<b>INS - Input from DX Port</b>	0110 110w
<b>INT n - Interrupt Type n</b>	1100 1101 : type
<b>INT - Single-Step Interrupt 3</b>	1100 1100
<b>INTO - Interrupt 4 on Overflow</b>	1100 1110
<b>INVD - Invalidate Cache</b>	0000 1111 : 0000 1000
<b>INVLPG - Invalidate TLB Entry</b>	0000 1111 : 0000 0001 : mod 111 r/m
<b>INVPCID - Invalidate Process-Context Identifier</b>	0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m
<b>IRET/IRETD - Interrupt Return</b>	1100 1111
<b>Jcc - Jump if Condition is Met</b>	
8-bit displacement	0111 tttn : 8-bit displacement
full displacement	0000 1111 : 1000 tttn : full displacement
<b>JCXZ/JECXZ - Jump on CX/ECX Zero</b> Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
<b>JMP - Unconditional Jump (to same segment)</b>	
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
<b>JMP - Unconditional Jump (to other segment)</b>	
direct intersegment	1110 1010 : unsigned full offset, selector
indirect intersegment	1111 1111 : mod 101 r/m
<b>LAHF - Load Flags into AHRegister</b>	1001 1111
<b>LAR - Load Access Rights Byte</b>	
from register	0000 1111 : 0000 0010 : 11 reg1 reg2
from memory	0000 1111 : 0000 0010 : mod reg r/m
<b>LDS - Load Pointer to DS</b>	1100 0101 : mod <sup>A,B</sup> reg r/m
<b>LEA - Load Effective Address</b>	1000 1101 : mod <sup>A</sup> reg r/m
<b>LEAVE - High Level Procedure Exit</b>	1100 1001
<b>LES - Load Pointer to ES</b>	1100 0100 : mod <sup>A,B</sup> reg r/m
<b>LFS - Load Pointer to FS</b>	0000 1111 : 1011 0100 : mod <sup>A</sup> reg r/m
<b>LGDT - Load Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 010 r/m

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>LGS - Load Pointer to GS</b>	0000 1111 : 1011 0101 : mod <sup>A</sup> reg r/m
<b>LIDT - Load Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 011 r/m
<b>LLDT - Load Local Descriptor Table Register</b>	
LDTR from register	0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0000 1111 : 0000 0000 : mod 010 r/m
<b>LMSW - Load Machine Status Word</b>	
from register	0000 1111 : 0000 0001 : 11 110 reg
from memory	0000 1111 : 0000 0001 : mod 110 r/m
<b>LOCK - Assert LOCK# Signal Prefix</b>	1111 0000
<b>LODS/LODSB/LODSW/LODSD - Load String Operand</b>	1010 110w
<b>LOOP - Loop Count</b>	1110 0010 : 8-bit displacement
<b>LOOPZ/LOOPE - Loop Count while Zero/Equal</b>	1110 0001 : 8-bit displacement
<b>LOOPNZ/LOOPNE - Loop Count while not Zero/Equal</b>	1110 0000 : 8-bit displacement
<b>LSL - Load Segment Limit</b>	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from memory	0000 1111 : 0000 0011 : mod reg r/m
<b>LSS - Load Pointer to SS</b>	0000 1111 : 1011 0010 : mod <sup>A</sup> reg r/m
<b>LTR - Load Task Register</b>	
from register	0000 1111 : 0000 0000 : 11 011 reg
from memory	0000 1111 : 0000 0000 : mod 011 r/m
<b>MOV - Move Data</b>	
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
AL, AX, or EAX to memory	1010 001w : full displacement
<b>MOV - Move to/from Control Registers</b>	
CR0 from register	0000 1111 : 0010 0010 : -- 000 reg
CR2 from register	0000 1111 : 0010 0010 : -- 010 reg
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
<b>MOV - Move to/from Debug Registers</b>	
DRO-DR3 from register	0000 1111 : 0010 0011 : -- eee reg

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
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
<b>MOV - Move to/from Segment Registers</b>	
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
<b>MOVBE - Move data after swapping bytes</b>	
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
<b>MOVS/MOVSB/MOVSW/MOVSD - Move Data from String to String</b>	1010 010w
<b>MOVSX - Move with Sign-Extend</b>	
memory to reg	0000 1111 : 1011 111w : mod reg r/m
<b>MOVZX - Move with Zero-Extend</b>	
register2 to register1	0000 1111 : 1011 011w : 11 reg1 reg2
memory to register	0000 1111 : 1011 011w : mod reg r/m
<b>MUL - Unsigned Multiply</b>	
AL, AX, or EAX with register	1111 011w : 11 100 reg
AL, AX, or EAX with memory	1111 011w : mod 100 r/m
<b>NEG - Two's Complement Negation</b>	
register	1111 011w : 11 011 reg
memory	1111 011w : mod 011 r/m
<b>NOP - No Operation</b>	1001 0000
<b>NOP - Multi-byte No Operation<sup>1</sup></b>	
register	0000 1111 0001 1111 : 11 000 reg
memory	0000 1111 0001 1111 : mod 000 r/m
<b>NOT - One's Complement Negation</b>	
register	1111 011w : 11 010 reg
memory	1111 011w : mod 010 r/m

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>OR - Logical Inclusive OR</b>	
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
<b>OUT - Output to Port</b>	
fixed port	1110 011w : port number
variable port	1110 111w
<b>OUTS - Output to DX Port</b>	0110 111w
<b>POP - Pop a Word from the Stack</b>	
register	1000 1111 : 11 000 reg
register (alternate encoding)	0101 1 reg
memory	1000 1111 : mod 000 r/m
<b>POP - Pop a Segment Register from the Stack</b> (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
<b>POPA/POPAD - Pop All General Registers</b>	0110 0001
<b>POPF/POFAD - Pop Stack into FLAGS or EFLAGS Register</b>	1001 1101
<b>PUSH - Push Operand onto the Stack</b>	
register	1111 1111 : 11 110 reg
register (alternate encoding)	0101 0 reg
memory	1111 1111 : mod 110 r/m
immediate	0110 10s0 : immediate data
<b>PUSH - Push Segment Register onto the Stack</b>	
segment register CS,DS,ES,SS	000 sreg2 110
segment register FS,GS	0000 1111: 10 sreg3 000
<b>PUSHA/PUSHAD - Push All General Registers</b>	0110 0000
<b>PUSHF/PUSHFD - Push Flags Register onto the Stack</b>	1001 1100
<b>RCL - Rotate thru Carry Left</b>	
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

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
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
<b>RCR - Rotate thru Carry Right</b>	
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
<b>RDMSR - Read from Model-Specific Register</b>	0000 1111 : 0011 0010
<b>RDPMC - Read Performance Monitoring Counters</b>	0000 1111 : 0011 0011
<b>RDTSC - Read Time-Stamp Counter</b>	0000 1111 : 0011 0001
<b>RDTSCP - Read Time-Stamp Counter and Processor ID</b>	0000 1111 : 0000 0001: 1111 1001
<b>REP INS - Input String</b>	1111 0011 : 0110 110w
<b>REP LODS - Load String</b>	1111 0011 : 1010 110w
<b>REP MOVS - Move String</b>	1111 0011 : 1010 010w
<b>REP OUTS - Output String</b>	1111 0011 : 0110 111w
<b>REP STOS - Store String</b>	1111 0011 : 1010 101w
<b>REPE CMPS - Compare String</b>	1111 0011 : 1010 011w
<b>REPE SCAS - Scan String</b>	1111 0011 : 1010 111w
<b>REPNE CMPS - Compare String</b>	1111 0010 : 1010 011w
<b>REPNE SCAS - Scan String</b>	1111 0010 : 1010 111w
<b>RET - Return from Procedure (to same segment)</b>	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
<b>RET - Return from Procedure (to other segment)</b>	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement
<b>ROL - Rotate Left</b>	
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

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>ROR - Rotate Right</b>	
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
<b>RSM - Resume from System Management Mode</b>	0000 1111 : 1010 1010
<b>SAHF - Store AH into Flags</b>	1001 1110
<b>SAL - Shift Arithmetic Left</b>	same instruction as SHL
<b>SAR - Shift Arithmetic Right</b>	
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
<b>SBB - Integer Subtraction with Borrow</b>	
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
<b>SCAS/SCASB/SCASW/SCASD - Scan String</b>	1010 111w
<b>SETcc - Byte Set on Condition</b>	
register	0000 1111 : 1001 tttn : 11 000 reg
memory	0000 1111 : 1001 tttn : mod 000 r/m
<b>SGDT - Store Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 000 r/m
<b>SHL - Shift Left</b>	
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

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>SHLD - Double Precision Shift Left</b>	
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
<b>SHR - Shift Right</b>	
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
<b>SHRD - Double Precision Shift Right</b>	
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
<b>SIDT - Store Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 001 r/m
<b>SLDT - Store Local Descriptor Table Register</b>	
to register	0000 1111 : 0000 0000 : 11 000 reg
to memory	0000 1111 : 0000 0000 : mod 000 r/m
<b>SMSW - Store Machine Status Word</b>	
to register	0000 1111 : 0000 0001 : 11 100 reg
to memory	0000 1111 : 0000 0001 : mod 100 r/m
<b>STC - Set Carry Flag</b>	1111 1001
<b>STD - Set Direction Flag</b>	1111 1101
<b>STI - Set Interrupt Flag</b>	1111 1011
<b>STOS/STOSB/STOSW/STOSD - Store String Data</b>	1010 101w
<b>STR - Store Task Register</b>	
to register	0000 1111 : 0000 0000 : 11 001 reg
to memory	0000 1111 : 0000 0000 : mod 001 r/m
<b>SUB - Integer Subtraction</b>	
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

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
immediate to memory	1000 00sw : mod 101 r/m : immediate data
<b>TEST - Logical Compare</b>	
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
<b>UD2 - Undefined instruction</b>	0000 FFFF : 0000 1011
<b>VERR - Verify a Segment for Reading</b>	
register	0000 1111 : 0000 0000 : 11 100 reg
memory	0000 1111 : 0000 0000 : mod 100 r/m
<b>VERW - Verify a Segment for Writing</b>	
register	0000 1111 : 0000 0000 : 11 101 reg
memory	0000 1111 : 0000 0000 : mod 101 r/m
<b>WAIT - Wait</b>	1001 1011
<b>WBINVD - Writeback and Invalidate Data Cache</b>	0000 1111 : 0000 1001
<b>WRMSR - Write to Model-Specific Register</b>	0000 1111 : 0011 0000
<b>XADD - Exchange and Add</b>	
register1, register2	0000 1111 : 1100 000w : 11 reg2 reg1
memory, reg	0000 1111 : 1100 000w : mod reg r/m
<b>XCHG - Exchange Register/Memory with Register</b>	
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
<b>XLAT/XLATB - Table Look-up Translation</b>	1101 0111
<b>XOR - Logical Exclusive OR</b>	
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
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
<b>Prefix Bytes</b>	
address size	0110 0111
LOCK	1111 0000
operand size	0110 0110
CS segment override	0010 1110

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
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 has no effect.

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode**

Instruction and Format	Encoding
<b>ADC - ADD with Carry</b>	
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
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
<b>ADD - Add</b>	
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

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister1 to qwordregister2	0100 1ROB 0000 0010 : 11 qwordreg1 qwordreg2
memory to register	0100 ORXB : 0000 001w : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0000 0000 : mod qwordreg r/m
register to memory	0100 ORXB : 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
<b>AND - Logical AND</b>	
register1 to register2	0100 OROB 0010 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1ROB 0010 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 OROB 0010 001w : 11 reg1 reg2
register1 to register2	0100 1ROB 0010 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 ORXB 0010 001w : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0010 0011 : mod qwordreg r/m
register to memory	0100 ORXB : 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
<b>BSF - Bit Scan Forward</b>	
register1, register2	0100 OROB 0000 1111 : 1011 1100 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1ROB 0000 1111 : 1011 1100 : 11 qwordreg1 qwordreg2
memory, register	0100 ORXB 0000 1111 : 1011 1100 : mod reg r/m
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1100 : mod qwordreg r/m
<b>BSR - Bit Scan Reverse</b>	
register1, register2	0100 OROB 0000 1111 : 1011 1101 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1ROB 0000 1111 : 1011 1101 : 11 qwordreg1 qwordreg2
memory, register	0100 ORXB 0000 1111 : 1011 1101 : mod reg r/m

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1101 : mod qwordreg r/m
<b>BSWAP - Byte Swap</b>	0000 1111 : 1100 1 reg
BSWAP - Byte Swap	0100 100B 0000 1111 : 1100 1 qwordreg
<b>BT - Bit Test</b>	
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 0ROB 0000 1111 : 1010 0011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1ROB 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
<b>BTC - Bit Test and Complement</b>	
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 0ROB 0000 1111 : 1011 1011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1ROB 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
<b>BTR - Bit Test and Reset</b>	
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 0ROB 0000 1111 : 1011 0011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1ROB 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
<b>BTS - Bit Test and Set</b>	
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 0ROB 0000 1111 : 1010 1011 : 11 reg2 reg1

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister1, qwordregister2	0100 1ROB 0000 1111 : 1010 1011 : 11 qwordreg2 qwordreg1
memory, register	0100 ORXB 0000 1111 : 1010 1011 : mod reg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1010 1011 : mod qwordreg r/m
<b>CALL - Call Procedure (in same segment)</b>	
direct	1110 1000 : displacement32
register indirect	0100 WR00 <sup>w</sup> 1111 1111 : 11 010 reg
memory indirect	0100 W0XB <sup>w</sup> 1111 1111 : mod 010 r/m
<b>CALL - Call Procedure (in other segment)</b>	
indirect	1111 1111 : mod 011 r/m
indirect	0100 10XB 0100 1000 1111 1111 : mod 011 r/m
<b>CBW - Convert Byte to Word</b>	1001 1000
<b>CDQ - Convert Doubleword to Qword+</b>	1001 1001
CDQE - RAX, Sign-Extend of EAX	0100 1000 1001 1001
<b>CLC - Clear Carry Flag</b>	1111 1000
<b>CLD - Clear Direction Flag</b>	1111 1100
<b>CLI - Clear Interrupt Flag</b>	1111 1010
<b>CLTS - Clear Task-Switched Flag in CR0</b>	0000 1111 : 0000 0110
<b>CMC - Complement Carry Flag</b>	1111 0101
<b>CMP - Compare Two Operands</b>	
register1 with register2	0100 OROB 0011 100w : 11 reg1 reg2
qwordregister1 with qwordregister2	0100 1ROB 0011 1001 : 11 qwordreg1 qwordreg2
register2 with register1	0100 OROB 0011 101w : 11 reg1 reg2
qwordregister2 with qwordregister1	0100 1ROB 0011 101w : 11 qwordreg1 qwordreg2
memory with register	0100 ORXB 0011 100w : mod reg r/m
memory64 with qwordregister	0100 1RXB 0011 1001 : mod qwordreg r/m
register with memory	0100 ORXB 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
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
<b>CMPS/CMPSB/CMPSW/CMPSD/CMPSQ - Compare String Operands</b>	
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

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>CMPXCHG - Compare and Exchange</b>	
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
<b>CPUID - CPU Identification</b>	0000 1111 : 1010 0010
CQO - Sign-Extend RAX	0100 1000 1001 1001
<b>CWD - Convert Word to Doubleword</b>	1001 1001
<b>CWDE - Convert Word to Doubleword</b>	1001 1000
<b>DEC - Decrement by 1</b>	
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
<b>DIV - Unsigned Divide</b>	
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
<b>ENTER - Make Stack Frame for High Level Procedure</b>	1100 1000 : 16-bit displacement : 8-bit level (L)
<b>HLT - Halt</b>	1111 0100
<b>IDIV - Signed Divide</b>	
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
<b>IMUL - Signed Multiply</b>	
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 ORXB 0000 1111 : 1010 1111 : mod reg r/m
qwordregister <- qwordregister withmemory64	0100 1RXB 0000 1111 : 1010 1111 : mod qwordreg r/m
register1 with immediate to register2	0100 OROB 0110 10s1 : 11 reg1 reg2 : imm

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister1 <- qwordregister2 with sign-extended immediate8	0100 1ROB 0110 1011 : 11 qwordreg1 qwordreg2 : imm8
qwordregister1 <- qwordregister2 with immediate32	0100 1ROB 0110 1001 : 11 qwordreg1 qwordreg2 : imm32
memory with immediate to register	0100 ORXB 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
<b>IN - Input From Port</b>	
fixed port	1110 010w : port number
variable port	1110 110w
<b>INC - Increment by 1</b>	
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
<b>INS - Input from DX Port</b>	0110 110w
<b>INT n - Interrupt Type n</b>	1100 1101 : type
<b>INT - Single-Step Interrupt 3</b>	1100 1100
<b>INTO - Interrupt 4 on Overflow</b>	1100 1110
<b>INVD - Invalidate Cache</b>	0000 1111 : 0000 1000
<b>INVLPG - Invalidate TLB Entry</b>	0000 1111 : 0000 0001 : mod 111 r/m
<b>INVPCID - Invalidate Process-Context Identifier</b>	0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m
<b>IRETO - Interrupt Return</b>	1100 1111
<b>Jcc - Jump if Condition is Met</b>	
8-bit displacement	0111 tttn : 8-bit displacement
displacements (excluding 16-bit relative offsets)	0000 1111 : 1000 tttn : displacement32
<b>JCXZ/JECXZ - Jump on CX/ECX Zero</b>	
Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
<b>JMP - Unconditional Jump (to same segment)</b>	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : displacement32
register indirect	0100 W00B <sup>w</sup> : 1111 1111 : 11 100 reg
memory indirect	0100 W0XB <sup>w</sup> : 1111 1111 : mod 100 r/m

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>JMP - Unconditional Jump (to other segment)</b>	
indirect intersegment	0100 00XB : 1111 1111 : mod 101 r/m
64-bit indirect intersegment	0100 10XB : 1111 1111 : mod 101 r/m
<b>LAR - Load Access Rights Byte</b>	
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 ORXB : 0000 1111 : 0000 0010 : mod reg r/m
from memory32 to qwordregister, masked by 00FxFF00H	0100 WRXB 0000 1111 : 0000 0010 : mod r/m
<b>LEA - Load Effective Address</b>	
in wordregister/dwordregister	0100 ORXB : 1000 1101 : mod <sup>A</sup> reg r/m
in qwordregister	0100 1RXB : 1000 1101 : mod <sup>A</sup> qwordreg r/m
<b>LEAVE - High Level Procedure Exit</b>	1100 1001
<b>LFS - Load Pointer to FS</b>	
FS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0100 : mod <sup>A</sup> reg r/m
FS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0100 : mod <sup>A</sup> qwordreg r/m
<b>LGDT - Load Global Descriptor Table Register</b>	0100 10XB : 0000 1111 : 0000 0001 : mod <sup>A</sup> 010 r/m
<b>LGS - Load Pointer to GS</b>	
GS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0101 : mod <sup>A</sup> reg r/m
GS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0101 : mod <sup>A</sup> qwordreg r/m
<b>LIDT - Load Interrupt Descriptor Table Register</b>	0100 10XB : 0000 1111 : 0000 0001 : mod <sup>A</sup> 011 r/m
<b>LLDT - Load Local Descriptor Table Register</b>	
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
<b>LMSW - Load Machine Status Word</b>	
from register	0100 000B : 0000 1111 : 0000 0001 : 11 110 reg
from memory	0100 00XB : 0000 1111 : 0000 0001 : mod 110 r/m
<b>LOCK - Assert LOCK# Signal Prefix</b>	1111 0000
<b>LODS/LODSB/LODSW/LOSD/LODSQ - Load String Operand</b>	
at DS:(E)SI to AL/EAX/EAX	1010 110w
at (RSI) to RAX	0100 1000 1010 1101
<b>LOOP - Loop Count</b>	
if count != 0, 8-bit displacement	1110 0010
if count != 0, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0010
<b>LOOPE - Loop Count while Zero/Equal</b>	
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

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>LOOPNE/LOOPNZ - Loop Count while not Zero/Equal</b>	
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
<b>LSL - Load Segment Limit</b>	
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
<b>LSS - Load Pointer to SS</b>	
SS:r16/r32 with far pointer from memory	0100 ORXB : 0000 1111 : 1011 0010 : mod <sup>A</sup> reg r/m
SS:r64 with far pointer from memory	0100 1WXB : 0000 1111 : 1011 0010 : mod <sup>A</sup> qwordreg r/m
<b>LTR - Load Task Register</b>	
from register	0100 ORO0 : 0000 1111 : 0000 0000 : 11 011 reg
from memory	0100 00XB : 0000 1111 : 0000 0000 : mod 011 r/m
<b>MOV - Move Data</b>	
register1 to register2	0100 OROB : 1000 100w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1ROB 1000 1001 : 11 qwordeg1 qwordreg2
register2 to register1	0100 OROB : 1000 101w : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1ROB 1000 1011 : 11 qwordreg1 qwordreg2
memory to reg	0100 ORXB : 1000 101w : mod reg r/m
memory64 to qwordregister	0100 1RXB 1000 1011 : mod qwordreg r/m
reg to memory	0100 ORXB : 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
<b>MOV - Move to/from Control Registers</b>	
CRO-CR4 from register	0100 OROB : 0000 1111 : 0010 0010 : 11 eee reg (eee = CR#)
CRx from qwordregister	0100 1ROB : 0000 1111 : 0010 0010 : 11 eee qwordreg (Reee = CR#)
register from CRO-CR4	0100 OROB : 0000 1111 : 0010 0000 : 11 eee reg (eee = CR#)

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister from CRx	0100 1ROB 0000 1111 : 0010 0000 : 11 eee qwordreg (Reee = CR#)
<b>MOV - Move to/from Debug Registers</b>	
DRO-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
DRO-DR7 from quadregister	0100 100B 0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
register from DRO-DR7	0000 1111 : 0010 0001 : 11 eee reg (eee = DR#)
quadregister from DRO-DR7	0100 100B 0000 1111 : 0010 0001 : 11 eee quadreg (eee = DR#)
<b>MOV - Move to/from Segment Registers</b>	
register to segment register	0100 W00B <sup>w</sup> : 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
<b>MOVBE - Move data after swapping bytes</b>	
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
<b>MOVS/MOVSB/MOVSW/MOVSQ - Move Data from String to String</b>	
Move data from string to string	1010 010w
Move data from string to string (qword)	0100 1000 1010 0101
<b>MOVSX/MOVSDX - Move with Sign-Extend</b>	
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
<b>MOVZX - Move with Zero-Extend</b>	
register2 to register1	0100 0R0B : 0000 1111 : 1011 011w : 11 reg1 reg2
dwordregister2 to qwordregister1	0100 1R0B 0000 1111 : 1011 0111 : 11 qwordreg1 dwordreg2

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
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
<b>MUL - Unsigned Multiply</b>	
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
<b>NEG - Two's Complement Negation</b>	
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
<b>NOP - No Operation</b>	1001 0000
<b>NOT - One's Complement Negation</b>	
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 10XB 1111 0111 : mod 010 r/m
<b>OR - Logical Inclusive OR</b>	
register1 to register2	0000 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 OROB 0000 1000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1ROB 0000 1001 : 11 qwordreg1 qwordreg2
register2 to register1	0000 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 OROB 0000 1010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 OROB 0000 1011 : 11 qwordreg1 qwordreg2
memory to register	0000 101w : mod reg r/m
memory8 to byteregister	0100 ORXB 0000 1010 : mod bytereg r/m
memory8 to qwordregister	0100 ORXB 0000 1011 : mod qwordreg r/m
register to memory	0000 100w : mod reg r/m
byteregister to memory8	0100 ORXB 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
immediate64 to RAX	0100 1000 0000 1101 : imm64
immediate to memory	1000 00sw : mod 001 r/m : imm

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
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
<b>OUT - Output to Port</b>	
fixed port	1110 011w : port number
variable port	1110 111w
<b>OUTS - Output to DX Port</b>	
output to DX Port	0110 111w
<b>POP - Pop a Value from the Stack</b>	
wordregister	0101 0101 : 0100 000B : 1000 1111 : 11 000 reg16
qwordregister	0100 W00B <sup>S</sup> : 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 <sup>S</sup> : 1000 1111 : mod 000 r/m
memory16	0101 0101 : 0100 00XB 1000 1111 : mod 000 r/m
<b>POP - Pop a Segment Register from the Stack</b> (Note: CS cannot be sreg2 in this usage.)	
segment register FS, GS	0000 1111: 10 sreg3 001
<b>POPF/POPFQ - Pop Stack into FLAGS/RFLAGS Register</b>	
pop stack to FLAGS register	0101 0101 : 1001 1101
pop Stack to RFLAGS register	0100 1000 1001 1101
<b>PUSH - Push Operand onto the Stack</b>	
wordregister	0101 0101 : 0100 000B : 1111 1111 : 11 110 reg16
qwordregister	0100 W00B <sup>S</sup> : 1111 1111 : 11 110 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 0 reg16
qwordregister (alternate encoding)	0100 W00B <sup>S</sup> : 0101 0 reg64
memory16	0101 0101 : 0100 000B : 1111 1111 : mod 110 r/m
memory64	0100 W00B <sup>S</sup> : 1111 1111 : mod 110 r/m
immediate8	0110 1010 : imm8
immediate16	0101 0101 : 0110 1000 : imm16
immediate64	0110 1000 : imm64
<b>PUSH - Push Segment Register onto the Stack</b>	
segment register FS,GS	0000 1111: 10 sreg3 000
<b>PUSHF/PUSHFD - Push Flags Register onto the Stack</b>	1001 1100
<b>RCL - Rotate thru Carry Left</b>	
register by 1	0100 000B : 1101 000w : 11 010 reg
qwordregister by 1	0100 100B 1101 0001 : 11 010 qwordreg
memory by 1	0100 00XB : 1101 000w : mod 010 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 010 r/m

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
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
<b>RCR - Rotate thru Carry Right</b>	
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
<b>RDMSR - Read from Model-Specific Register</b>	
load ECX-specified register into EDX:EAX	0000 1111 : 0011 0010
<b>RDPMC - Read Performance Monitoring Counters</b>	
load ECX-specified performance counter into EDX:EAX	0000 1111 : 0011 0011
<b>RDTSC - Read Time-Stamp Counter</b>	
read time-stamp counter into EDX:EAX	0000 1111 : 0011 0001
<b>RDTSCP - Read Time-Stamp Counter and Processor ID</b>	0000 1111 : 0000 0001: 1111 1001
<b>REP INS - Input String</b>	
<b>REP LODS - Load String</b>	
<b>REP MOVS - Move String</b>	
<b>REP OUTS - Output String</b>	
<b>REP STOS - Store String</b>	
<b>REPE CMPS - Compare String</b>	
<b>REPE SCAS - Scan String</b>	
<b>REPNE CMPS - Compare String</b>	
<b>REPNE SCAS - Scan String</b>	
<b>RET - Return from Procedure (to same segment)</b>	

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
<b>RET - Return from Procedure (to other segment)</b>	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement
<b>ROL - Rotate Left</b>	
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
<b>ROR - Rotate Right</b>	
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
memory64 by CL	0100 10XB 1101 0011 : mod 001 r/m
register by immediate count	0100 000B 1100 000w : 11 001 reg : imm8

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
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
<b>RSM - Resume from System Management Mode</b>	0000 1111 : 1010 1010
<b>SAL - Shift Arithmetic Left</b>	same instruction as SHL
<b>SAR - Shift Arithmetic Right</b>	
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
<b>SBB - Integer Subtraction with Borrow</b>	
register1 to register2	0100 OROB 0001 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 OROB 0001 1000 : 11 bytereg1 bytereg2
quadregister1 to quadregister2	0100 1ROB 0001 1001 : 11 quadreg1 quadreg2
register2 to register1	0100 OROB 0001 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 OROB 0001 1010 : 11 reg1 bytereg2
byteregister2 to byteregister1	0100 1ROB 0001 1011 : 11 reg1 bytereg2
memory to register	0100 ORXB 0001 101w : mod reg r/m
memory8 to byteregister	0100 ORXB 0001 1010 : mod bytereg r/m
memory64 to byteregister	0100 1RXB 0001 1011 : mod quadreg r/m
register to memory	0100 ORXB 0001 100w : mod reg r/m
byteregister to memory8	0100 ORXB 0001 1000 : mod reg r/m

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
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
<b>SCAS/SCASB/SCASW/SCASD - Scan String</b>	
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
<b>SETcc - Byte Set on Condition</b>	
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
<b>SGDT - Store Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 000 r/m
<b>SHL - Shift Left</b>	
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
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

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
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
<b>SHLD - Double Precision Shift Left</b>	
register by immediate count	0100 0ROB 0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1ROB 0000 1111 : 1010 0100 : 11 qwordreg2 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 0ROB 0000 1111 : 1010 0101 : 11 reg2 reg1
quadregister by CL	0100 1ROB 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
<b>SHR - Shift Right</b>	
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
<b>SHRD - Double Precision Shift Right</b>	
register by immediate count	0100 0ROB 0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1ROB 0000 1111 : 1010 1100 : 11 qwordreg2 qwordreg1 : imm8
memory by immediate count	0100 0RXB 0000 1111 : 1010 1100 : mod reg r/m : imm8
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

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister by CL	0100 1ROB 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
<b>SIDT - Store Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 001 r/m
<b>SLDT - Store Local Descriptor Table Register</b>	
to register	0100 000B 0000 1111 : 0000 0000 : 11 000 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 000 r/m
<b>SMSW - Store Machine Status Word</b>	
to register	0100 000B 0000 1111 : 0000 0001 : 11 100 reg
to memory	0100 00XB 0000 1111 : 0000 0001 : mod 100 r/m
<b>STC - Set Carry Flag</b>	1111 1001
<b>STD - Set Direction Flag</b>	1111 1101
<b>STI - Set Interrupt Flag</b>	1111 1011
<b>STOS/STOSB/STOSW/STOSD/STOSQ - Store String Data</b>	
store string data	1010 101w
store string data (RAX at address RDI)	0100 1000 1010 1011
<b>STR - Store Task Register</b>	
to register	0100 000B 0000 1111 : 0000 0000 : 11 001 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 001 r/m
<b>SUB - Integer Subtraction</b>	
register1 from register2	0100 0R0B 0010 100w : 11 reg1 reg2
byteregister1 from byteregister2	0100 0R0B 0010 1000 : 11 bytereg1 bytereg2
qwordregister1 from qwordregister2	0100 1ROB 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 1ROB 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
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

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
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
<b>SWAPGS - Swap GS Base Register</b>	
Exchanges the current GS base register value for value in MSR C0000102H	0000 1111 0000 0001 1111 1000
<b>SYSCALL - Fast System Call</b>	
fast call to privilege level 0 system procedures	0000 1111 0000 0101
<b>SYSRET - Return From Fast System Call</b>	
return from fast system call	0000 1111 0000 0111
<b>TEST - Logical Compare</b>	
register1 and register2	0100 OROB 1000 010w : 11 reg1 reg2
byteregister1 and byteregister2	0100 OROB 1000 0100 : 11 bytereg1 bytereg2
qwordregister1 and qwordregister2	0100 1ROB 1000 0101 : 11 qwordreg1 qwordreg2
memory and register	0100 OROB 1000 010w : mod reg r/m
memory8 and byteregister	0100 ORXB 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
<b>UD2 - Undefined instruction</b>	0000 FFFF : 0000 1011
<b>VERR - Verify a Segment for Reading</b>	
register	0100 000B 0000 1111 : 0000 0000 : 11 100 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 100 r/m
<b>VERW - Verify a Segment for Writing</b>	
register	0100 000B 0000 1111 : 0000 0000 : 11 101 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 101 r/m
<b>WAIT - Wait</b>	1001 1011
<b>WBINVD - Writeback and Invalidate Data Cache</b>	0000 1111 : 0000 1001
<b>WRMSR - Write to Model-Specific Register</b>	
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

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>XADD - Exchange and Add</b>	
register1, register2	0100 0ROB 0000 1111 : 1100 000w : 11 reg2 reg1
byteregister1, byteregister2	0100 0ROB 0000 1111 : 1100 0000 : 11 bytereg2 bytereg1
qwordregister1, qwordregister2	0100 0ROB 0000 1111 : 1100 0001 : 11 qwordreg2 qwordreg1
memory, register	0100 ORXB 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
<b>XCHG - Exchange Register/Memory with Register</b>	
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
<b>XLAT/XLATB - Table Look-up Translation</b>	
AL to byte DS:[(E)BX + unsigned AL]	1101 0111
AL to byte DS:[RBX + unsigned AL]	0100 1000 1101 0111
<b>XOR - Logical Exclusive OR</b>	
register1 to register2	0100 ORXB 0011 000w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0ROB 0011 0000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1ROB 0011 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0ROB 0011 001w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0ROB 0011 0010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 1ROB 0011 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 ORXB 0011 001w : mod reg r/m
memory8 to byteregister	0100 ORXB 0011 0010 : mod bytereg r/m
memory64 to qwordregister	0100 1RXB 0011 0011 : mod qwordreg r/m
register to memory	0100 ORXB 0011 000w : mod reg r/m
byteregister to memory8	0100 ORXB 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
immediate32 to memory64	0100 10XB 1000 0001 : mod 110 r/m : imm32
immediate8 to memory64	0100 10XB 1000 0011 : mod 110 r/m : imm8

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>Prefix Bytes</b>	
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
<b>CMPXCHG8B - Compare and Exchange 8 Bytes</b>	
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
<b>CMPXCHG8B/CMPXCHG16B - Compare and Exchange Bytes</b>	
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
<b>EMMS - Empty MMX technology state</b>	0000 1111:01110111
<b>MOVD - Move doubleword</b>	
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
<b>MOVQ - Move quadword</b>	
mmxreg2 to mmxreg1	0000 1111:0110 1111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:0111 1111: 11 mmxreg1 mmxreg2
mem to mmxreg	0000 1111:0110 1111: mod mmxreg r/m

**Table B-19. MMX Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
mem from mmxreg	0000 1111:0111 1111: mod mmxreg r/m
<b>PACKSSDW<sup>1</sup> - Pack dword to word data (signed with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 1011:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 1011: mod mmxreg r/m
<b>PACKSSWB<sup>1</sup> - Pack word to byte data (signed with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 0011:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0011: mod mmxreg r/m
<b>PACKUSWB<sup>1</sup> - Pack word to byte data (unsigned with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 0111:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0111: mod mmxreg r/m
<b>PADD - Add with wrap-around</b>	
mmxreg2 to mmxreg1	0000 1111:1111 11gg:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1111 11gg: mod mmxreg r/m
<b>PADDS - Add signed with saturation</b>	
mmxreg2 to mmxreg1	0000 1111:1110 11gg:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 11gg: mod mmxreg r/m
<b>PADDUS - Add unsigned with saturation</b>	
mmxreg2 to mmxreg1	0000 1111:1101 11gg:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 11gg: mod mmxreg r/m
<b>PAND - Bitwise And</b>	
mmxreg2 to mmxreg1	0000 1111:1101 1011:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1011: mod mmxreg r/m
<b>PANDN - Bitwise AndNot</b>	
mmxreg2 to mmxreg1	0000 1111:1101 1111:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1111: mod mmxreg r/m
<b>PCMPEQ - Packed compare for equality</b>	
mmxreg1 with mmxreg2	0000 1111:0111 01gg:11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0111 01gg: mod mmxreg r/m
<b>PCMPGT - Packed compare greater (signed)</b>	
mmxreg1 with mmxreg2	0000 1111:0110 01gg:11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0110 01gg: mod mmxreg r/m
<b>PMADDWD - Packed multiply add</b>	
mmxreg2 to mmxreg1	0000 1111:1111 0101:11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1111 0101: mod mmxreg r/m
<b>PMULHUW - Packed multiplication, store high word (unsigned)</b>	
mmxreg2 to mmxreg1	0000 1111:1110 0100:11 mmxreg1 mmxreg2

**Table B-19. MMX Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
memory to mmxreg	0000 1111:1110 0100: mod mmxreg r/m
<b>PMULHW - Packed multiplication, store high word</b>	
mmxreg2 to mmxreg1	0000 1111:1110 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 0101: mod mmxreg r/m
<b>PMULLW - Packed multiplication, store low word</b>	
mmxreg2 to mmxreg1	0000 1111:1101 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 0101: mod mmxreg r/m
<b>POR - Bitwise Or</b>	
mmxreg2 to mmxreg1	0000 1111:1110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1011: mod mmxreg r/m
<b>PSLL<sup>2</sup> - Packed shift left logical</b>	
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
<b>PSRA<sup>2</sup> - Packed shift right arithmetic</b>	
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
<b>PSRL<sup>2</sup> - Packed shift right logical</b>	
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
<b>PSUB - Subtract with wrap-around</b>	
mmxreg2 from mmxreg1	0000 1111:1111 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1111 10gg: mod mmxreg r/m
<b>PSUBS - Subtract signed with saturation</b>	
mmxreg2 from mmxreg1	0000 1111:1110 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1110 10gg: mod mmxreg r/m
<b>PSUBUS - Subtract unsigned with saturation</b>	
mmxreg2 from mmxreg1	0000 1111:1101 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1101 10gg: mod mmxreg r/m
<b>PUNPCKH - Unpack high data to next larger type</b>	
mmxreg2 to mmxreg1	0000 1111:0110 10gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 10gg: mod mmxreg r/m
<b>PUNPCKL - Unpack low data to next larger type</b>	
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
<b>PXOR - Bitwise Xor</b>	
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
<b>XGETBV - Get Value of Extended Control Register</b>	0000 1111:0000 0001: 1101 0000
<b>XRSTOR - Restore Processor Extended States<sup>7</sup></b>	0000 1111:1010 1110: mod <sup>A</sup> 101 r/m
<b>XSAVE - Save Processor Extended States<sup>7</sup></b>	0000 1111:1010 1110: mod <sup>A</sup> 100 r/m
<b>XSETBV - Set Extended Control Register</b>	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
<b>CMOVcc - Conditional Move</b>	
register2 to register1	0000 1111: 0100 tttn : 11 reg1 reg2
memory to register	0000 1111 : 0100 tttn : mod reg r/m
<b>FCMOVcc - Conditional Move on EFLAG Register Condition Codes</b>	
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)

**Table B-21. Formats and Encodings of P6 Family Instructions (Contd.)**

Instruction and Format	Encoding
move if not below or equal (NBE)	11011 011 : 11 010 ST(i)
move if not unordered (NU)	11011 011 : 11 011 ST(i)
FCOMI - Compare Real and Set EFLAGS	11011 011 : 11 110 ST(i)
<b>FXRSTOR - Restore x87 FPU, MMX, SSE, and SSE2 State<sup>7</sup></b>	0000 1111:1010 1110: mod <sup>A</sup> 001 r/m
<b>FXSAVE - Save x87 FPU, MMX, SSE, and SSE2 State<sup>7</sup></b>	0000 1111:1010 1110: mod <sup>A</sup> 000 r/m
<b>SYSENTER - Fast System Call</b>	0000 1111:0011 0100
<b>SYSEXIT - Fast Return from Fast System Call</b>	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
<b>ADDPS—Add Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1000: mod xmmreg r/m
<b>ADDSS—Add Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01011000: mod xmmreg r/m
<b>ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0101: mod xmmreg r/m
<b>ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0100: mod xmmreg r/m
<b>CMPPS—Compare Packed Single-Precision Floating-Point Values</b>	
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
<b>CMPSS—Compare Scalar Single-Precision Floating-Point Values</b>	

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
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
<b>COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 to xmmreg1	0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1111: mod xmmreg r/m
<b>CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</b>	
mmreg to xmmreg	0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0000 1111:0010 1010: mod xmmreg r/m
<b>CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1101: mod mmreg r/m
<b>CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</b>	
r32 to xmmreg1	1111 0011:0000 1111:00101010:11 xmmreg1 r32
mem to xmmreg	1111 0011:0000 1111:00101010: mod xmmreg r/m
<b>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
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
<b>CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0000 1111:0010 1100:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1100: mod mmreg r/m
<b>CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
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
<b>DIVPS—Divide Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1110: mod xmmreg r/m
<b>DIVSS—Divide Scalar Single-Precision Floating-Point Values</b>	
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
<b>LDMXCSR—Load MXCSR Register State</b>	
m32 to MXCSR	0000 1111:1010 1110:mod <sup>A</sup> 010 mem
<b>MAXPS—Return Maximum Packed Single-Precision Floating-Point Values</b>	

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1111: mod xmmreg r/m
<b>MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value</b>	
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
<b>MINPS—Return Minimum Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1101: mod xmmreg r/m
<b>MINSS—Return Minimum Scalar Double-Precision Floating-Point Value</b>	
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
<b>MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values</b>	
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
<b>MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low</b>	
xmmreg2 to xmmreg1	0000 1111:0001 0010:11 xmmreg1 xmmreg2
<b>MOVHPS—Move High Packed Single-Precision Floating-Point Values</b>	
mem to xmmreg	0000 1111:0001 0110: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0111: mod xmmreg r/m
<b>MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High</b>	
xmmreg2 to xmmreg1	0000 1111:00010110:11 xmmreg1 xmmreg2
<b>MOVLPS—Move Low Packed Single-Precision Floating-Point Values</b>	
mem to xmmreg	0000 1111:0001 0010: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0011: mod xmmreg r/m
<b>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	0000 1111:0101 0000:11 r32 xmmreg
<b>MOVSS—Move Scalar Single-Precision Floating-Point Values</b>	
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

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
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
<b>MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values</b>	
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
<b>MULPS—Multiply Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1001: mod xmmreg r/m
<b>MULSS—Multiply Scalar Single-Precision Floating-Point Values</b>	
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
<b>ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0110: mod xmmreg r/m
<b>RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0011: mod xmmreg r/m
<b>RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01010011: mod xmmreg r/m
<b>RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0010: mode xmmreg r/m
<b>RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value</b>	
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
<b>SHUFPS—Shuffle Packed Single-Precision Floating-Point Values</b>	
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
<b>SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values</b>	

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0001: mod xmmreg r/m
<b>SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value</b>	
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
<b>STMXCSR—Store MXCSR Register State</b>	
MXCSR to mem	0000 1111:1010 1110:mod <sup>A</sup> 011 mem
<b>SUBPS—Subtract Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1100:mod xmmreg r/m
<b>SUBSS—Subtract Scalar Single-Precision Floating-Point Values</b>	
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
<b>UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 to xmmreg1	0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1110: mod xmmreg r/m
<b>UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0101: mod xmmreg r/m
<b>UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0100: mod xmmreg r/m
<b>XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0111:11 xmmreg1 xmmreg2
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
<b>PAVGB/PAVGW—Average Packed Integers</b>	
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
<b>PEXTRW—Extract Word</b>	
mmreg to reg32, imm8	0000 1111:1100 0101:11 r32 mmreg: imm8
<b>PINSRW—Insert Word</b>	
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
<b>PMAXSW—Maximum of Packed Signed Word Integers</b>	
mmreg2 to mmreg1	0000 1111:1110 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1110: mod mmreg r/m
<b>PMAXUB—Maximum of Packed Unsigned Byte Integers</b>	
mmreg2 to mmreg1	0000 1111:1101 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1110: mod mmreg r/m
<b>PMINSW—Minimum of Packed Signed Word Integers</b>	
mmreg2 to mmreg1	0000 1111:1110 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1010: mod mmreg r/m
<b>PMINUB—Minimum of Packed Unsigned Byte Integers</b>	
mmreg2 to mmreg1	0000 1111:1101 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1010: mod mmreg r/m
<b>PMOVMSKB—Move Byte Mask To Integer</b>	
mmreg to reg32	0000 1111:1101 0111:11 r32 mmreg
<b>PMULHUW—Multiply Packed Unsigned Integers and Store High Result</b>	
mmreg2 to mmreg1	0000 1111:1110 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0100: mod mmreg r/m
<b>PSADBW—Compute Sum of Absolute Differences</b>	
mmreg2 to mmreg1	0000 1111:1111 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0110: mod mmreg r/m
<b>PSHUFW—Shuffle Packed Words</b>	
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
<b>MASKMOVQ—Store Selected Bytes of Quadword</b>	
mmreg2 to mmreg1	0000 1111:1111 0111:11 mmreg1 mmreg2
<b>MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</b>	
xmmreg to mem	0000 1111:0010 1011: mod xmmreg r/m
<b>MOVNTQ—Store Quadword Using Non-Temporal Hint</b>	
mmreg to mem	0000 1111:1110 0111: mod mmreg r/m
<b>PREFETCHT0—Prefetch Temporal to All Cache Levels</b>	0000 1111:0001 1000:mod <sup>A</sup> 001 mem
<b>PREFETCHT1—Prefetch Temporal to First Level Cache</b>	0000 1111:0001 1000:mod <sup>A</sup> 010 mem
<b>PREFETCHT2—Prefetch Temporal to Second Level Cache</b>	0000 1111:0001 1000:mod <sup>A</sup> 011 mem
<b>PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels</b>	0000 1111:0001 1000:mod <sup>A</sup> 000 mem
<b>SFENCE—Store Fence</b>	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
<b>ADDPD—Add Packed Double-Precision Floating-Point Values</b>	
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
<b>ADDSD—Add Scalar Double-Precision Floating-Point Values</b>	
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
<b>ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</b>	
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
<b>ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values</b>	
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
<b>CMPPD—Compare Packed Double-Precision Floating-Point Values</b>	
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
<b>CMPSD—Compare Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1, imm8	1111 0010:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0110:0000 1111:1100 0010: mod xmmreg r/m; imm8
<b>COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
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
<b>CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</b>	
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
<b>CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
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
<b>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</b>	
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
<b>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
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
<b>CVTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
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
<b>CVTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	1111 0010:0000 1111:0010 1100:11 r32 xmmreg
mem to r32	1111 0010:0000 1111:0010 1100: mod r32 r/m
<b>CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values</b>	
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
<b>CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1010: mod xmmreg r/m
<b>CVTSD2SS—Covert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value</b>	
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
<b>CVTSS2SD—Covert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:00001 1111:0101 1010: mod xmmreg r/m
<b>CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
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
<b>CVTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
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

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</b>	
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
<b>CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
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
<b>CVTTPS2DQ—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
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
<b>CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1011: mod xmmreg r/m
<b>DIVPD—Divide Packed Double-Precision Floating-Point Values</b>	
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
<b>DIVSD—Divide Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1110: mod xmmreg r/m
<b>MAXPD—Return Maximum Packed Double-Precision Floating-Point Values</b>	
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
<b>MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value</b>	
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
<b>MINPD—Return Minimum Packed Double-Precision Floating-Point Values</b>	
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
<b>MINSD—Return Minimum Scalar Double-Precision Floating-Point Value</b>	
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

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values</b>	
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
<b>MOVHPD—Move High Packed Double-Precision Floating-Point Values</b>	
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
<b>MOVLPD—Move Low Packed Double-Precision Floating-Point Values</b>	
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
<b>MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	0110 0110:0000 1111:0101 0000:11 r32 xmmreg
<b>MOVSD—Move Scalar Double-Precision Floating-Point Values</b>	
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
<b>MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values</b>	
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
<b>MULPD—Multiply Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1001: mod xmmreg r/m
<b>MULSD—Multiply Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:00001111:01011001: mod xmmreg r/m
<b>ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
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

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>SHUFPD—Shuffle Packed Double-Precision Floating-Point Values</b>	
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
<b>SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values</b>	
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
<b>SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value</b>	
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
<b>SUBPD—Subtract Packed Double-Precision Floating-Point Values</b>	
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
<b>SUBSD—Subtract Scalar Double-Precision Floating-Point Values</b>	
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
<b>UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
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
<b>UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values</b>	
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
<b>UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values</b>	
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
<b>XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
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
<b>MOVD—Move Doubleword</b>	
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
<b>MOVDQA—Move Aligned Double Quadword</b>	
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
<b>MOVDQU—Move Unaligned Double Quadword</b>	
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
<b>MOVQ2DQ—Move Quadword from MMX to XMM Register</b>	
mmreg to xmmreg	1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2
<b>MOVDQ2Q—Move Quadword from XMM to MMX Register</b>	
xmmreg to mmreg	1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2
<b>MOVQ—Move Quadword</b>	
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
<b>PACKSSDW<sup>1</sup>—Pack Dword To Word Data (signed with saturation)</b>	
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
<b>PACKSSWB—Pack Word To Byte Data (signed with saturation)</b>	
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
<b>PACKUSWB—Pack Word To Byte Data (unsigned with saturation)</b>	
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
<b>PADDQ—Add Packed Quadword Integers</b>	
mmreg2 to mmreg1	0000 1111:1101 0100:11 mmreg1 mmreg2

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
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
<b>PADD—Add With Wrap-around</b>	
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
<b>PADDS—Add Signed With Saturation</b>	
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
<b>PADDUS—Add Unsigned With Saturation</b>	
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
<b>PAND—Bitwise And</b>	
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
<b>PANDN—Bitwise AndNot</b>	
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
<b>PAVGB—Average Packed Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:11100 000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100000 mod xmmreg r/m
<b>PAVGW—Average Packed Integers</b>	
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
<b>PCMPEQ—Packed Compare For Equality</b>	
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
<b>PCMPGT—Packed Compare Greater (signed)</b>	
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
<b>PEXTRW—Extract Word</b>	
xmmreg to reg32, imm8	0110 0110:0000 1111:1100 0101:11 r32 xmmreg; imm8
<b>PINSRW—Insert Word</b>	
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
<b>PMADDWD—Packed Multiply Add</b>	
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

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
<b>PMAXSW—Maximum of Packed Signed Word Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110: mod xmmreg r/m
<b>PMAXUB—Maximum of Packed Unsigned Byte Integers</b>	
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
<b>PMINSW—Minimum of Packed Signed Word Integers</b>	
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
<b>PMINUB—Minimum of Packed Unsigned Byte Integers</b>	
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
<b>PMOVMSKB—Move Byte Mask To Integer</b>	
xmmreg to reg32	0110 0110:0000 1111:1101 0111:11 r32 xmmreg
<b>PMULHUW—Packed multiplication, store high word (unsigned)</b>	
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
<b>PMULHW—Packed Multiplication, store high word</b>	
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
<b>PMULLW—Packed Multiplication, store low word</b>	
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
<b>PMULUDQ—Multiply Packed Unsigned Doubleword Integers</b>	
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
<b>POR—Bitwise Or</b>	
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
<b>PSADBW—Compute Sum of Absolute Differences</b>	
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
<b>PSHUFLW—Shuffle Packed Low Words</b>	

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
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
<b>PSHUFHW—Shuffle Packed High Words</b>	
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
<b>PSHUFD—Shuffle Packed Doublewords</b>	
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
<b>PSLLDQ—Shift Double Quadword Left Logical</b>	
xmmreg, imm8	0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8
<b>PSLL—Packed Shift Left Logical</b>	
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
<b>PSRA—Packed Shift Right Arithmetic</b>	
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
<b>PSRLDQ—Shift Double Quadword Right Logical</b>	
xmmreg, imm8	0110 0110:00001111:01110011:11 0111 xmmreg: imm8
<b>PSRL—Packed Shift Right Logical</b>	
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
<b>PSUBQ—Subtract Packed Quadword Integers</b>	
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
<b>PSUB—Subtract With Wrap-around</b>	
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
<b>PSUBS—Subtract Signed With Saturation</b>	
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
<b>PSUBUS—Subtract Unsigned With Saturation</b>	
xmmreg2 from xmmreg1	0000 1111:1101 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:1101 10gg: mod xmmreg r/m

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
<b>PUNPCKH—Unpack High Data To Next Larger Type</b>	
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
<b>PUNPCKHQDQ—Unpack High Data</b>	
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
<b>PUNPCKL—Unpack Low Data To Next Larger Type</b>	
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
<b>PUNPCKLQDQ—Unpack Low Data</b>	
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
<b>PXOR—Bitwise Xor</b>	
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
<b>MASKMOVDQU—Store Selected Bytes of Double Quadword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0111:11 xmmreg1 xmmreg2
<b>CLFLUSH—Flush Cache Line</b>	
mem	0000 1111:1010 1110: mod 111 r/m
<b>MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</b>	
xmmreg to mem	0110 0110:0000 1111:0010 1011: mod xmmreg r/m
<b>MOVNTDQ—Store Double Quadword Using Non-Temporal Hint</b>	
xmmreg to mem	0110 0110:0000 1111:1110 0111: mod xmmreg r/m
<b>MOVNTI—Store Doubleword Using Non-Temporal Hint</b>	
reg to mem	0000 1111:1100 0011: mod reg r/m
<b>PAUSE—Spin Loop Hint</b>	1111 0011:1001 0000
<b>LFENCE—Load Fence</b>	0000 1111:1010 1110: 11 101 000
<b>MFENCE—Memory Fence</b>	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
<b>ADDSUBPD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010000: mod xmmreg r/m
<b>ADDSUBPS—Add /Sub packed SP FP numbers from XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	11110010:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11010000: mod xmmreg r/m
<b>HADDPD—Add horizontally packed DP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111100: mod xmmreg r/m
<b>HADDPSP—Add horizontally packed SP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	11110010:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111100: mod xmmreg r/m
<b>HSUBPD—Sub horizontally packed DP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111101: mod xmmreg r/m
<b>HSUBPS—Sub horizontally packed SP FP numbers XMM2/Mem to XMM1</b>	
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
<b>MONITOR—Set up a linear address range to be monitored by hardware</b>	
eax, ecx, edx	0000 1111 : 0000 0001:11 001 000
<b>MWAIT—Wait until write-back store performed within the range specified by the instruction MONITOR</b>	
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
<b>FISTTP—Store ST in int16 (chop) and pop</b>	
m16int	11011 111 : mod <sup>A</sup> 001 r/m
<b>FISTTP—Store ST in int32 (chop) and pop</b>	
m32int	11011 011 : mod <sup>A</sup> 001 r/m
<b>FISTTP—Store ST in int64 (chop) and pop</b>	
m64int	11011 101 : mod <sup>A</sup> 001 r/m
<b>LDDQU—Load unaligned integer 128-bit</b>	
xmm, m128	11110010:00001111:11110000: mod <sup>A</sup> xmmreg r/m
<b>MOVDDUP—Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate</b>	
xmmreg2 to xmmreg1	11110010:00001111:00010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:00010010: mod xmmreg r/m
<b>MOVSHDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high</b>	
xmmreg2 to xmmreg1	11110011:00001111:00010110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010110: mod xmmreg r/m
<b>MOVSLDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low</b>	
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
<b>PABSB—Packed Absolute Value Bytes</b>	
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
<b>PABSD—Packed Absolute Value Double Words</b>	
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
<b>PABSW—Packed Absolute Value Words</b>	

**Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)**

Instruction and Format	Encoding
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
<b>PALIGNR—Packed Align Right</b>	
mmreg2 to mmreg1, imm8	0000 1111:0011 1010: 0000 1111:11 mmreg1 mmreg2: imm8
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
<b>PHADDD—Packed Horizontal Add Double Words</b>	
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
<b>PHADDSW—Packed Horizontal Add and Saturate</b>	
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
<b>PHADDW—Packed Horizontal Add Words</b>	
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
<b>PHSUBD—Packed Horizontal Subtract Double Words</b>	
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
<b>PHSUBSW—Packed Horizontal Subtract and Saturate</b>	
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

**Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)**

Instruction and Format	Encoding
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
<b>PHSUBW—Packed Horizontal Subtract Words</b>	
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
mem to xmmreg	0110 0110:0000 1111:0011 1000:0000 0101: mod xmmreg r/m
<b>PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes</b>	
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
<b>PMULHRSW—Packed Multiply Hign with Round and Scale</b>	
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
<b>PSHUFB—Packed Shuffle Bytes</b>	
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
<b>PSIGNB—Packed Sign Bytes</b>	
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
<b>PSIGND—Packed Sign Double Words</b>	
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
<b>PSIGNW—Packed Sign Words</b>	

**Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)**

Instruction and Format	Encoding
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
<b>AESDEC—Perform One Round of an AES Decryption Flow</b>	
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
<b>AESDECLAST—Perform Last Round of an AES Decryption Flow</b>	
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
<b>AESENC—Perform One Round of an AES Encryption Flow</b>	
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
<b>AESENCLAST—Perform Last Round of an AES Encryption Flow</b>	
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
<b>AESIMC—Perform the AES InvMixColumn Transformation</b>	
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
<b>AESKEYGENASSIST—AES Round Key Generation Assist</b>	
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
<b>PCLMULQDQ—Carry-Less Multiplication Quadword</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010:0100 0100:11 xmmreg1 xmmreg2: imm8

**Table B-33. Formats and Encodings of AESNI and PCLMULQDQ Instructions**

Instruction and Format	Encoding
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
<b>CMOVcc—Conditional Move</b>	
register2 to register1	0100 0ROB 0000 1111:0100 tttn : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1ROB 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
<b>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0ROB 1111 0010:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1ROB 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
<b>CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</b>	
r32 to xmmreg1	0100 0ROB 1111 0011:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1ROB 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
<b>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</b>	
r32 to xmmreg1	0100 0ROB 1111 0010:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1ROB 1111 0010:0000 1111:0010 1010:11 xmmreg r64
mem to xmmreg	0100 0RXB 1111 0010:0000 1111:0010 1010: mod xmmreg r/m

**Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)**

Instruction and Format	Encoding
mem64 to xmmreg	0100 1RXB 1111 0010:0000 1111:0010 1010: mod xmmreg r/m
<b>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0ROB 1111 0011:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1ROB 1111 0011:0000 1111:0010 1101:11 r64 xmmreg
mem to r32	0100 ORXB 11110011:00001111:00101101: mod r32 r/m
mem32 to r64	0100 1RXB 1111 0011:0000 1111:0010 1101: mod r64 r/m
<b>CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0ROB 11110010:00001111:00101100:11 r32 xmmreg
xmmreg to r64	0100 1ROB 1111 0010:0000 1111:0010 1100:11 r64 xmmreg
mem64 to r32	0100 ORXB 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
<b>CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0ROB 1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
xmmreg to r64	0100 1ROB 1111 0011:0000 1111:0010 1100:11 r64 xmmreg1
mem to r32	0100 ORXB 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
<b>MOVD/MOVQ—Move doubleword</b>	
reg to mmxreg	0100 0ROB 0000 1111:0110 1110:11 mmxreg reg
qwordreg to mmxreg	0100 1ROB 0000 1111:0110 1110:11 mmxreg qwordreg
reg from mmxreg	0100 0ROB 0000 1111:0111 1110:11 mmxreg reg
qwordreg from mmxreg	0100 1ROB 0000 1111:0111 1110:11 mmxreg qwordreg
mem to mmxreg	0100 ORXB 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 ORXB 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 ORXB 0000 1111:0110 01gg: mod mmxreg r/m
<b>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	0100 0ROB 0000 1111:0101 0000:11 r32 xmmreg
xmmreg to r64	0100 1ROB 00001111:01010000:11 r64 xmmreg
<b>PEXTRW—Extract Word</b>	
mmreg to reg32, imm8	0100 0ROB 0000 1111:1100 0101:11 r32 mmreg:imm8

**Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)**

Instruction and Format	Encoding
mmreg to reg64, imm8	0100 1ROB 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 1ROB 0110 0110 0000 1111:1100 0101:11 r64 xmmreg: imm8
<b>PINSRW—Insert Word</b>	
reg32 to mmreg, imm8	0100 0R0B 0000 1111:1100 0100:11 mmreg r32: imm8
reg64 to mmreg, imm8	0100 1ROB 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
<b>PMOVMSKB—Move Byte Mask To Integer</b>	
mmreg to reg32	0100 0RXB 0000 1111:1101 0111:11 r32 mmreg
mmreg to reg64	0100 1ROB 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 require 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
<b>BLENDPD — Blend Packed Double-Precision Floats</b>	
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
<b>BLENDPS — Blend Packed Single-Precision Floats</b>	
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
<b>BLENDVPD — Variable Blend Packed Double-Precision Floats</b>	

**Table B-35. Encodings of SSE4.1 instructions**

<b>Instruction and Format</b>	<b>Encoding</b>
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
<b>BLENDVPS – Variable Blend Packed Single-Precision Floats</b>	
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
<b>DPPD – Packed Double-Precision Dot Products</b>	
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
<b>DPPS – Packed Single-Precision Dot Products</b>	
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
<b>EXTRACTPS – Extract From Packed Single-Precision Floats</b>	
reg from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111:11 xmmreg reg: imm8
mem from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111: mod xmmreg r/m: imm8
<b>INSERTPS – Insert Into Packed Single-Precision Floats</b>	
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
<b>MOVNTDQA – Load Double Quadword Non-temporal Aligned</b>	
m128 to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1010:11 r/m xmmreg2
<b>MPSADBW – Multiple Packed Sums of Absolute Difference</b>	
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
<b>PACKUSDW – Pack with Unsigned Saturation</b>	
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
<b>PBLENDVB – Variable Blend Packed Bytes</b>	
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

**Table B-35. Encodings of SSE4.1 instructions**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>PBLENDW – Blend Packed Words</b>	
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
<b>PCMPEQQ – Compare Packed Qword Data of Equal</b>	
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
<b>PEXTRB – Extract Byte</b>	
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
<b>PEXTRD – Extract DWord</b>	
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
<b>PEXTRQ – Extract QWord</b>	
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
<b>PEXTRW – Extract Word</b>	
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
<b>PHMINPOSUW – Packed Horizontal Word Minimum</b>	
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
<b>PINSRB – Extract Byte</b>	
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
<b>PINSRD – Extract DWord</b>	
reg to xmmreg, imm8	0110 0110:0000 1111:0011 1010:0010 0010:11 xmmreg reg: imm8

**Table B-35. Encodings of SSE4.1 instructions**

<b>Instruction and Format</b>	<b>Encoding</b>
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010:0010 0010: mod xmmreg r/m; imm8
<b>PINSRQ – Extract QWord</b>	
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
<b>PMAXSB – Maximum of Packed Signed Byte Integers</b>	
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
<b>PMAXSD – Maximum of Packed Signed Dword Integers</b>	
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
<b>PMAXUD – Maximum of Packed Unsigned Dword Integers</b>	
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
<b>PMAXUW – Maximum of Packed Unsigned Word Integers</b>	
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
<b>PMINSB – Minimum of Packed Signed Byte Integers</b>	
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
<b>PMINSD – Minimum of Packed Signed Dword Integers</b>	
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
<b>PMINUD – Minimum of Packed Unsigned Dword Integers</b>	
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
<b>PMINUW – Minimum of Packed Unsigned Word Integers</b>	
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
<b>PMOVSXBD – Packed Move Sign Extend - Byte to Dword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:0010 0001:11 xmmreg1 xmmreg2

**Table B-35. Encodings of SSE4.1 instructions**

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0001: mod xmmreg r/m
<b>PMOVSBQ — Packed Move Sign Extend - Byte to Qword</b>	
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
<b>PMOVSBW — Packed Move Sign Extend - Byte to Word</b>	
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
<b>PMOVSXWD — Packed Move Sign Extend - Word to Dword</b>	
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
<b>PMOVSXWQ — Packed Move Sign Extend - Word to Qword</b>	
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
<b>PMOVSDQ — Packed Move Sign Extend - Dword to Qword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0101: mod xmmreg r/m
<b>PMOVZXBD — Packed Move Zero Extend - Byte to Dword</b>	
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
<b>PMOVZXBQ — Packed Move Zero Extend - Byte to Qword</b>	
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
<b>PMOVZXBW — Packed Move Zero Extend - Byte to Word</b>	
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
<b>PMOVZXWD — Packed Move Zero Extend - Word to Dword</b>	
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
<b>PMOVZXWQ — Packed Move Zero Extend - Word to Qword</b>	
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
<b>PMOVZXDQ — Packed Move Zero Extend - Dword to Qword</b>	

**Table B-35. Encodings of SSE4.1 instructions**

Instruction and Format	Encoding
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
<b>PMULDQ – Multiply Packed Dword Integers</b>	
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
<b>PMULLD – Multiply Packed Dword Integers, Store low Result</b>	
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
<b>PTEST – Logical Compare</b>	
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
<b>ROUNDPD – Round Packed Double-Precision Values</b>	
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
<b>ROUNDPS – Round Packed Single-Precision Values</b>	
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
<b>ROUNDSD – Round Scalar Double-Precision Value</b>	
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
<b>ROUNDSS – Round Scalar Single-Precision Value</b>	
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 require REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

**Table B-36. Encodings of SSE4.2 instructions**

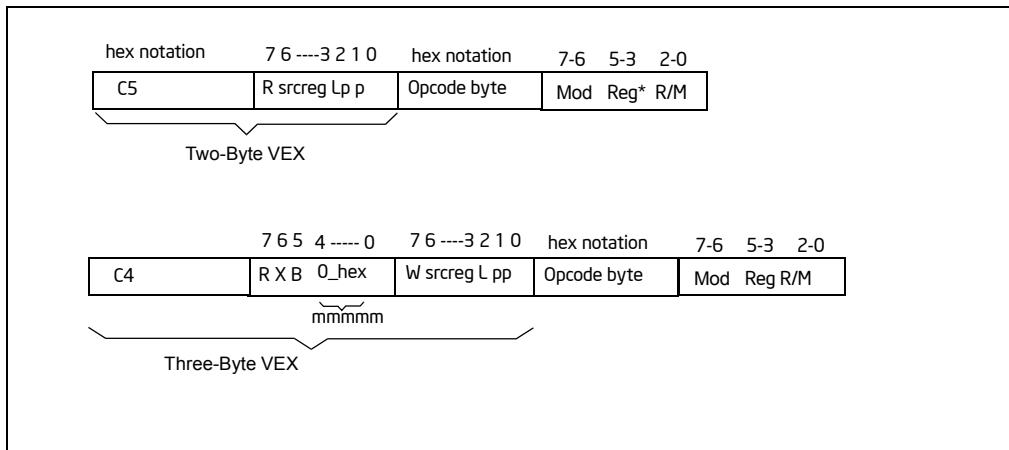
<b>Instruction and Format</b>	<b>Encoding</b>
<b>CRC32 – Accumulate CRC32</b>	
reg2 to reg1	1111 0010:0000 1111:0011 1000: 1111 000w :11 reg1 reg2
mem to reg	1111 0010:0000 1111:0011 1000: 1111 000w : mod reg r/m
bytereg2 to reg1	1111 0010:0100 WROB:0000 1111:0011 1000: 1111 0000 :11 reg1 bytereg2
m8 to reg	1111 0010:0100 WROB:0000 1111:0011 1000: 1111 0000 : mod reg r/m
qwreg2 to qwreg1	1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0000 :11 qwreg1 qwreg2
mem64 to qwreg	1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0000 : mod qwreg r/m
<b>PCMPESTRI— Packed Compare Explicit-Length Strings To Index</b>	
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
<b>PCMPESTRM— Packed Compare Explicit-Length Strings To Mask</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0000: mod xmmreg r/m
<b>PCMPISTRI— Packed Compare Implicit-Length String To Index</b>	
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
<b>PCMPISTRM— Packed Compare Implicit-Length Strings To Mask</b>	
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
<b>PCMPGTQ— Packed Compare Greater Than</b>	
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
<b>POPCNT— Return Number of Bits Set to 1</b>	
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
<b>VBLENDPD – Blend Packed Double-Precision Floats</b>	
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
<b>VBLENDPS – Blend Packed Single-Precision Floats</b>	
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
<b>VBLENDVPD – Variable Blend Packed Double-Precision Floats</b>	
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
<b>VBLENDVPS – Variable Blend Packed Single-Precision Floats</b>	

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
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
<b>VDPPD – Packed Double-Precision Dot Products</b>	
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
<b>VDPPS – Packed Single-Precision Dot Products</b>	
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
<b>VEXTRACTPS – Extract From Packed Single-Precision Floats</b>	
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
<b>VINSERTPS – Insert Into Packed Single-Precision Floats</b>	
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
<b>VMOVNTDQA – Load Double Quadword Non-temporal Aligned</b>	
m128 to xmmreg1	C4: rxb0_2: w_F 001:2A:11 xmmreg1 r/m
<b>VMPSADBW – Multiple Packed Sums of Absolute Difference</b>	
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
<b>VPACKUSDW – Pack with Unsigned Saturation</b>	
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
<b>VPBLENDVB – Variable Blend Packed Bytes</b>	
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
<b>VPBLENDW – Blend Packed Words</b>	
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
<b>VPCMPEQQ – Compare Packed Qword Data of Equal</b>	
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:

Instruction and Format	Encoding
<b>VPEXTRB – Extract Byte</b>	
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
<b>VPEXTRD – Extract DWord</b>	
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
<b>VPEXTRQ – Extract QWord</b>	
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
<b>VPEXTRW – Extract Word</b>	
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
<b>VPHMINPOSUW – Packed Horizontal Word Minimum</b>	
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
<b>VPINSRB – Insert Byte</b>	
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
<b>VPINSRD – Insert DWord</b>	
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
<b>VPINSRQ – Insert QWord</b>	
r64 with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 1 xmmreg2 001:22:11 xmmreg1 reg: imm
m64 with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 1 xmmreg2 001:22:mod xmmreg1 r/m: imm
<b>VPMAXSB – Maximum of Packed Signed Byte Integers</b>	
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
<b>VPMAXSD – Maximum of Packed Signed Dword Integers</b>	
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
<b>VPMAXUD – Maximum of Packed Unsigned Dword Integers</b>	
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
<b>VPMAXUW – Maximum of Packed Unsigned Word Integers</b>	
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
<b>VPMINSB – Minimum of Packed Signed Byte Integers</b>	
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

Instruction and Format	Encoding
<b>VPMINS – Minimum of Packed Signed Dword Integers</b>	
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
<b>VPMINUD – Minimum of Packed Unsigned Dword Integers</b>	
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
<b>VPMINUW – Minimum of Packed Unsigned Word Integers</b>	
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
<b>VPMOVSXBD – Packed Move Sign Extend - Byte to Dword</b>	
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
<b>VPMOVSBQ – Packed Move Sign Extend - Byte to Qword</b>	
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
<b>VPMOVSBW – Packed Move Sign Extend - Byte to Word</b>	
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
<b>VPMOVSXWD – Packed Move Sign Extend - Word to Dword</b>	
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
<b>VPMOVSXWQ – Packed Move Sign Extend - Word to Qword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:24:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:24:mod xmmreg1 r/m
<b>VPMOVSDQ – Packed Move Sign Extend - Dword to Qword</b>	
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
<b>VPMOVZXBD – Packed Move Zero Extend - Byte to Dword</b>	
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
<b>VPMOVZXBQ – Packed Move Zero Extend - Byte to Qword</b>	
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
<b>VPMOVZXBW – Packed Move Zero Extend - Byte to Word</b>	
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
<b>VPMOVZXWD – Packed Move Zero Extend - Word to Dword</b>	

Instruction and Format	Encoding
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
<b>VPMOVZXWQ — Packed Move Zero Extend - Word to Qword</b>	
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
<b>VPMOVZXDQ — Packed Move Zero Extend - Dword to Qword</b>	
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
<b>VMULDDQ — Multiply Packed Signed Dword Integers</b>	
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
<b>VMULLD — Multiply Packed Signed Dword Integers, Store low Result</b>	
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
<b>VPTEST — Logical Compare</b>	
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
yymmreg2 to yymmreg1	C4: rxb0_2: w_F 101:17:11 yymmreg1 yymmreg2
mem to yymmreg	C4: rxb0_2: w_F 101:17:mod yymmreg1 r/m
<b>VROUNDPD — Round Packed Double-Precision Values</b>	
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
yymmreg2 to yymmreg1, imm8	C4: rxb0_3: w_F 101:09:11 yymmreg1 yymmreg2: imm
mem to yymmreg1, imm8	C4: rxb0_3: w_F 101:09:mod yymmreg1 r/m: imm
<b>VROUNDPS — Round Packed Single-Precision Values</b>	
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
yymmreg2 to yymmreg1, imm8	C4: rxb0_3: w_F 101:08:11 yymmreg1 yymmreg2: imm
mem to yymmreg1, imm8	C4: rxb0_3: w_F 101:08:mod yymmreg1 r/m: imm
<b>VROUNDSD — Round Scalar Double-Precision Value</b>	
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
<b>VROUNDSS — Round Scalar Single-Precision Value</b>	
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

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
<b>VPCMPESTRI — Packed Compare Explicit Length Strings, Return Index</b>	
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
<b>VPCMPESTRM — Packed Compare Explicit Length Strings, Return Mask</b>	
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
<b>VPCMPGTQ — Compare Packed Data for Greater Than</b>	
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
<b>VPCMPISTRI — Packed Compare Implicit Length Strings, Return Index</b>	
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
<b>VPCMPISTRM — Packed Compare Implicit Length Strings, Return Mask</b>	
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
<b>VAESDEC — Perform One Round of an AES Decryption Flow</b>	
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
<b>VAESDECLAST — Perform Last Round of an AES Decryption Flow</b>	
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
<b>VAESENC — Perform One Round of an AES Encryption Flow</b>	
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
<b>VAESENCLAST — Perform Last Round of an AES Encryption Flow</b>	
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
<b>VAESIMC — Perform the AES InvMixColumn Transformation</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:DB:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:DB:mod xmmreg1 r/m
<b>VAESKEYGENASSIST — AES Round Key Generation Assist</b>	
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
<b>VPABSB — Packed Absolute Value</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:1C:11 xmmreg1 xmmreg2

Instruction and Format	Encoding
mem to xmmreg1	C4: rxb0_2: w_F 001:1C:mod xmmreg1 r/m
<b>VPABSD — Packed Absolute Value</b>	
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
<b>VPABSW — Packed Absolute Value</b>	
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
<b>VPALIGNR — Packed Align Right</b>	
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
<b>VPHADD — Packed Horizontal Add</b>	
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
<b>VPHADDW — Packed Horizontal Add</b>	
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
<b>VPHADDSW — Packed Horizontal Add and Saturate</b>	
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
<b>VPHSUBD — Packed Horizontal Subtract</b>	
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
<b>VPHSUBW — Packed Horizontal Subtract</b>	
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
<b>VPHSUBSW — Packed Horizontal Subtract and Saturate</b>	
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
<b>VPMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes</b>	
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
<b>VPMULHRSW — Packed Multiply High with Round and Scale</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:0B:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:0B:mod xmmreg1 r/m
<b>VPSHUFBB — Packed Shuffle Bytes</b>	
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
<b>VPSIGNB — Packed SIGN</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:08:11 xmmreg1 xmmreg3

<b>Instruction and Format</b>	<b>Encoding</b>
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:08:mod xmmreg1 r/m
<b>VPSIGND — Packed SIGN</b>	
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
<b>VPSIGNW — Packed SIGN</b>	
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
<b>VADDSUBPD — Packed Double-FP Add/Subtract</b>	
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 <sup>1</sup> 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
<b>VADDSUBPS — Packed Single-FP Add/Subtract</b>	
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
<b>VHADDPPD — Packed Double-FP Horizontal Add</b>	
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
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:7C:mod ymmreg1 r/m
<b>VHADDPS — Packed Single-FP Horizontal Add</b>	
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

Instruction and Format	Encoding
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
<b>VHSUBPD — Packed Double-FP Horizontal Subtract</b>	
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
<b>VHSUBPS — Packed Single-FP Horizontal Subtract</b>	
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
<b>VLDDQU — Load Unaligned Integer 128 Bits</b>	
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
<b>VMOVDDUP — Move One Double-FP and Duplicate</b>	
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
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
<b>VMOVHLPS — Move Packed Single-Precision Floating-Point Values High to Low</b>	
xmmreg2 and xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:12:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
xmmreglo2 and xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:12:11 xmmreg1 xmmreglo3
<b>VMOVSHDUP – Move Packed Single-FP High and Duplicate</b>	
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
<b>VMOVSLDUP – Move Packed Single-FP Low and Duplicate</b>	
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
<b>VADDPD – Add Packed Double-Precision Floating-Point Values</b>	
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
<b>VADDSD – Add Scalar Double-Precision Floating-Point Values</b>	
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
<b>VANDPD – Bitwise Logical AND of Packed Double-Precision Floating-Point Values</b>	
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

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:54:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:54:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:54:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:54:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:54:mod yymmreg1 r/m
<b>VANDNPD — Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</b>	
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
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:55:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:55:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:55:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:55:mod yymmreg1 r/m
<b>VCMPPD — Compare Packed Double-Precision Floating-Point Values</b>	
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
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:C2:11 yymmreg1 yymmreg3: imm
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:C2:mod yymmreg1 r/m: imm
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:C2:11 yymmreg1 yymmreglo3: imm
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:C2:mod yymmreg1 r/m: imm
<b>VCMPSD — Compare Scalar Double-Precision Floating-Point Values</b>	
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
<b>VCOMISD — Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
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
<b>VCVTQ2PD— Convert Packed Dword Integers to Packed Double-Precision FP Values</b>	
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

<b>Instruction and Format</b>	<b>Encoding</b>
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
<b>VCVTQD2PS— Convert Packed Dword Integers to Packed Single-Precision FP Values</b>	
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
<b>VCVTPD2DQ— Convert Packed Double-Precision FP Values to Packed Dword Integers</b>	
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
<b>VCVTPD2PS— Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values</b>	
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
<b>VCVTPS2DQ— Convert Packed Single-Precision FP Values to Packed Dword Integers</b>	
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

Instruction and Format	Encoding
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
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
<b>VCVTPS2PD— Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values</b>	
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
<b>VCVTSD2SI— Convert Scalar Double-Precision FP Value to Integer</b>	
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
<b>VCVTSD2SS — Convert Scalar Double-Precision FP Value to Scalar Single-Precision FP Value</b>	
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
<b>VCVTSI2SD— Convert Dword Integer to Scalar Double-Precision FP Value</b>	
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
<b>VCVTSS2SD — Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5A:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
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
<b>VCVTPD2DQ— Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers</b>	
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
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
<b>VCVTPS2DQ— Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers</b>	
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
<b>VCVTSD2SI— Convert with Truncation Scalar Double-Precision FP Value to Signed Integer</b>	
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
<b>VDIVPD — Divide Packed Double-Precision Floating-Point Values</b>	
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

Instruction and Format	Encoding
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5E:mod ymmreg1 r/m
<b>VDIVSD — Divide Scalar Double-Precision Floating-Point Values</b>	
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
<b>VMASKMOVDQU— Store Selected Bytes of Double Quadword</b>	
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
<b>VMAXPD — Return Maximum Packed Double-Precision Floating-Point Values</b>	
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
<b>VMAXSD — Return Maximum Scalar Double-Precision Floating-Point Value</b>	
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
<b>VMINPD — Return Minimum Packed Double-Precision Floating-Point Values</b>	
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
<b>VMINSD — Return Minimum Scalar Double-Precision Floating-Point Value</b>	
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

Instruction and Format	Encoding
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
<b>VMOVAPD — Move Aligned Packed Double-Precision Floating-Point Values</b>	
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
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
<b>VMOVD — Move Doubleword</b>	
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
<b>VMOVQ — Move Quadword</b>	
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
<b>VMOVEDQA — Move Aligned Double Quadword</b>	
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

Instruction and Format	Encoding
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 xmmreglo	C5: r_F 101:7F:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 101:7F:mod r/m ymmreg1
<b>VMOVDQU — Move Unaligned Double Quadword</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:6F:11 xmmreg1 xmmreg2
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
<b>VMOVHPD — Move High Packed Double-Precision Floating-Point Value</b>	
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
<b>VMOVLPD — Move Low Packed Double-Precision Floating-Point Value</b>	
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

Instruction and Format	Encoding
xmmreglo to mem	C5: r_F 001:13:mod r/m xmmreglo
<b>VMOVMSKPD — Extract Packed Double-Precision Floating-Point Sign Mask</b>	
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
<b>VMOVNTDQ — Store Double Quadword Using Non-Temporal Hint</b>	
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
<b>VMOVNTPD — Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</b>	
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:11 r/m ymmreg1
ymmreglo to mem	C5: r_F 101:2B:11 r/m ymmreglo
<b>VMOVSD — Move Scalar Double-Precision Floating-Point Value</b>	
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
<b>VMOVUPD — Move Unaligned Packed Double-Precision Floating-Point Values</b>	
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

Instruction and Format	Encoding
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
<b>VMULPD – Multiply Packed Double-Precision Floating-Point Values</b>	
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
<b>VMULSD – Multiply Scalar Double-Precision Floating-Point Values</b>	
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
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
<b>VORPD – Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
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
<b>VPACKSSW— Pack with Signed Saturation</b>	
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
<b>VPACKSSDW— Pack with Signed Saturation</b>	
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

<b>Instruction and Format</b>	<b>Encoding</b>
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
<b>VPACKUSWB— Pack with Unsigned Saturation</b>	
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
<b>VPADDB — Add Packed Integers</b>	
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
<b>VPADDW — Add Packed Integers</b>	
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
<b>VPADDD — Add Packed Integers</b>	
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
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
<b>VPADDQ — Add Packed Quadword Integers</b>	
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
<b>VPADDSS — Add Packed Signed Integers with Signed Saturation</b>	
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
<b>VPADDSSW — Add Packed Signed Integers with Signed Saturation</b>	
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

Instruction and Format	Encoding
<b>VPADDUSB – Add Packed Unsigned Integers with Unsigned Saturation</b>	
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
<b>VPADDUSW – Add Packed Unsigned Integers with Unsigned Saturation</b>	
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
<b>VPAND – Logical AND</b>	
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
<b>VPANDN – Logical AND NOT</b>	
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
<b>VPAVGB – Average Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E0:11 xmmreg1 xmmreg3
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
<b>VPAVGW – Average Packed Integers</b>	
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
<b>VPCMPEQB – Compare Packed Data for Equal</b>	
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
<b>VPCMPEQW – Compare Packed Data for Equal</b>	
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

Instruction and Format	Encoding
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
<b>VPCMPEQD – Compare Packed Data for Equal</b>	
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
<b>VPCMPGTB – Compare Packed Signed Integers for Greater Than</b>	
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
<b>VPCMPGTw – Compare Packed Signed Integers for Greater Than</b>	
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
<b>VPCMPGTD – Compare Packed Signed Integers for Greater Than</b>	
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
<b>VPEXTRW – Extract Word</b>	
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
<b>VPINSRW – Insert Word</b>	
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
<b>VPMADDWD – Multiply and Add Packed Integers</b>	
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
<b>VPMAXSW – Maximum of Packed Signed Word Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EE:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
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
<b>VPMAXUB – Maximum of Packed Unsigned Byte Integers</b>	
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
<b>VPMINSW – Minimum of Packed Signed Word Integers</b>	
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
<b>VPMINUB – Minimum of Packed Unsigned Byte Integers</b>	
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
<b>VPMOVMSKB – Move Byte Mask</b>	
xmmreg1 to reg	C4: rxb0_1: w_F 001:D7:11 reg xmmreg1
xmmreg1 to reg	C5: r_F 001:D7:11 reg xmmreg1
<b>VPMULHUW – Multiply Packed Unsigned Integers and Store High Result</b>	
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
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E4:mod xmmreg1 r/m
<b>VPMULHW – Multiply Packed Signed Integers and Store High Result</b>	
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
<b>VPMULLW – Multiply Packed Signed Integers and Store Low Result</b>	
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

Instruction and Format	Encoding
<b>VPMULUDQ — Multiply Packed Unsigned Doubleword Integers</b>	
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
<b>VPOR — Bitwise Logical OR</b>	
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
<b>VPSADBW — Compute Sum of Absolute Differences</b>	
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
<b>VPSHUFD — Shuffle Packed Doublewords</b>	
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
<b>VPSHUFHW — Shuffle Packed High Words</b>	
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
<b>VPSHUFLW — Shuffle Packed Low Words</b>	
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
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
<b>VPSLLDQ — Shift Double Quadword Left Logical</b>	
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
<b>VPSLLW — Shift Packed Data Left Logical</b>	
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

Instruction and Format	Encoding
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
<b>VPSLLD – Shift Packed Data Left Logical</b>	
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
<b>VPSLLQ – Shift Packed Data Left Logical</b>	
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
<b>VPSRAW – Shift Packed Data Right Arithmetic</b>	
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
<b>VPSRAD – Shift Packed Data Right Arithmetic</b>	
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
xmmreglo to xmmreg1 using imm8	C5: r_F 001:72:11 xmmreg1 xmmreglo: imm
<b>VPSRLDQ – Shift Double Quadword Right Logical</b>	
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
<b>VPSRLW – Shift Packed Data Right Logical</b>	
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

Instruction and Format	Encoding
xmmreglo to xmmreg1 using imm8	C5: r_F 001:71:11 xmmreg1 xmmreglo: imm
<b>VPSRLD — Shift Packed Data Right Logical</b>	
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
<b>VPSRLQ — Shift Packed Data Right Logical</b>	
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
<b>VPSUBB — Subtract Packed Integers</b>	
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
<b>VPSUBW — Subtract Packed Integers</b>	
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
<b>VPSUBD — Subtract Packed Integers</b>	
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
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FA:mod xmmreg1 r/m
<b>VPSUBQ — Subtract Packed Quadword Integers</b>	
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
<b>VPSUBSB — Subtract Packed Signed Integers with Signed Saturation</b>	
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

Instruction and Format	Encoding
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
<b>VPSUBSW – Subtract Packed Signed Integers with Signed Saturation</b>	
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
<b>VPSUBUSB – Subtract Packed Unsigned Integers with Unsigned Saturation</b>	
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
<b>VPSUBUSW – Subtract Packed Unsigned Integers with Unsigned Saturation</b>	
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
<b>VPUNPCKHBW – Unpack High Data</b>	
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
<b>VPUNPCKHWD – Unpack High Data</b>	
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
<b>VPUNPCKHDQ – Unpack High Data</b>	
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
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
<b>VPUNPCKHQDQ – Unpack High Data</b>	
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

Instruction and Format	Encoding
<b>VPUNPCKLBW – Unpack Low Data</b>	
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
<b>VPUNPCKLWD – Unpack Low Data</b>	
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
<b>VPUNPCKLDQ – Unpack Low Data</b>	
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
<b>VPUNPCKLQDQ – Unpack Low Data</b>	
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
<b>VPXOR – Logical Exclusive OR</b>	
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
<b>VSHUFPD – Shuffle Packed Double-Precision Floating-Point Values</b>	
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
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
<b>VSQRTPD – Compute Square Roots of Packed Double-Precision Floating-Point Values</b>	
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

Instruction and Format	Encoding
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
<b>VSQRTSD — Compute Square Root of Scalar Double-Precision Floating-Point Value</b>	
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
<b>VSUBPD — Subtract Packed Double-Precision Floating-Point Values</b>	
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
<b>VSUBSD — Subtract Scalar Double-Precision Floating-Point Values</b>	
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
<b>VUCOMISD — Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS</b>	
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
<b>VUNPCKHPD — Unpack and Interleave High Packed Double-Precision Floating-Point Values</b>	
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
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

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
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
<b>VUNPCKHPS — Unpack and Interleave High Packed Single-Precision Floating-Point Values</b>	
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
<b>VUNPCKLPD — Unpack and Interleave Low Packed Double-Precision Floating-Point Values</b>	
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
<b>VUNPCKLPS — Unpack and Interleave Low Packed Single-Precision Floating-Point Values</b>	
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
<b>VXORPD — Bitwise Logical XOR for Double-Precision Floating-Point Values</b>	
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
<b>VADDPS — Add Packed Single-Precision Floating-Point Values</b>	
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
<b>VADDSS — Add Scalar Single-Precision Floating-Point Values</b>	
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
<b>VANDPS — Bitwise Logical AND of Packed Single-Precision Floating-Point Values</b>	
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
<b>VANDNPS — Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values</b>	
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
<b>VCMPPS — Compare Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:C2:11 xmmreg1 xmmreg3: imm

Instruction and Format	Encoding
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
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
<b>VCMPPS — Compare Scalar Single-Precision Floating-Point Values</b>	
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
<b>VCOMISS — Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
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
<b>VCVTSI2SS — Convert Dword Integer to Scalar Single-Precision FP Value</b>	
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
<b>VCVTSS2SI — Convert Scalar Single-Precision FP Value to Dword Integer</b>	
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
<b>VCVTTSS2SI — Convert with Truncation Scalar Single-Precision FP Value to Dword Integer</b>	
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

Instruction and Format	Encoding
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
<b>VDIVPS – Divide Packed Single-Precision Floating-Point Values</b>	
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
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
<b>VDIVSS – Divide Scalar Single-Precision Floating-Point Values</b>	
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
<b>VLDMXCSR – Load MXCSR Register</b>	
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
<b>VMAXPS – Return Maximum Packed Single-Precision Floating-Point Values</b>	
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
<b>VMAXSS – Return Maximum Scalar Single-Precision Floating-Point Value</b>	
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
<b>VMINPS – Return Minimum Packed Single-Precision Floating-Point Values</b>	
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

<b>Instruction and Format</b>	<b>Encoding</b>
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
<b>VMINSS – Return Minimum Scalar Single-Precision Floating-Point Value</b>	
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
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
<b>VMOVAPS – Move Aligned Packed Single-Precision Floating-Point Values</b>	
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 xmmreglo	C5: r_F 100:29:11 xmmreglo ymmreg1
ymmreg1 to mem	C5: r_F 100:29:mod r/m ymmreg1
<b>VMOVHPS – Move High Packed Single-Precision Floating-Point Values</b>	
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
<b>VMOVLHPS – Move Packed Single-Precision Floating-Point Values Low to High</b>	
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

Instruction and Format	Encoding
<b>VMOVLPS — Move Low Packed Single-Precision Floating-Point Values</b>	
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
<b>VMOVMSKPS — Extract Packed Single-Precision Floating-Point Sign Mask</b>	
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
<b>VMOVNTPS — Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</b>	
xmmreg1 to mem	C4: rxb0_1: w_F 000:2B:mod r/m xmmreg1
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
<b>VMOVSS — Move Scalar Single-Precision Floating-Point Values</b>	
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
<b>VMOVUPS — Move Unaligned Packed Single-Precision Floating-Point Values</b>	
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

Instruction and Format	Encoding
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
<b>VMULPS — Multiply Packed Single-Precision Floating-Point Values</b>	
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
<b>VMULSS — Multiply Scalar Single-Precision Floating-Point Values</b>	
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
<b>VORPS — Bitwise Logical OR of Single-Precision Floating-Point Values</b>	
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
<b>VRCPPS — Compute Reciprocals of Packed Single-Precision Floating-Point Values</b>	
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

Instruction and Format	Encoding
mem to ymmreg1	C5: r_F 100:53:mod ymmreg1 r/m
<b>VRCPSS — Compute Reciprocal of Scalar Single-Precision Floating-Point Values</b>	
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
<b>VRSQRTPS — Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</b>	
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
<b>VRSQRTSS — Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value</b>	
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
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
<b>VSHUFPS — Shuffle Packed Single-Precision Floating-Point Values</b>	
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
<b>VSQRTPS — Compute Square Roots of Packed Single-Precision Floating-Point Values</b>	
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

Instruction and Format	Encoding
ymmreglo to ymmreg1	C5: r_F 100:51:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:51:mod ymmreg1 r/m
<b>VSQRTSS — Compute Square Root of Scalar Single-Precision Floating-Point Value</b>	
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
<b>VSTMXCSR — Store MXCSR Register State</b>	
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
<b>VSUBPS — Subtract Packed Single-Precision Floating-Point Values</b>	
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
<b>VSUBSS — Subtract Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5C:11 xmmreg1 xmmreg3
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
<b>VUCOMISS — Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS</b>	
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
<b>UNPCKHPS — Unpack and Interleave High Packed Single-Precision Floating-Point Values</b>	
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
<b>UNPCKLPS — Unpack and Interleave Low Packed Single-Precision Floating-Point Value</b>	

Instruction and Format	Encoding
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
<b>VXORPS – Bitwise Logical XOR for Single-Precision Floating-Point Values</b>	
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
<b>VBROADCAST – Load with Broadcast</b>	
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
<b>VEXTRACTF128 – Extract Packed Floating-Point Values</b>	
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
<b>VINSERTF128 – Insert Packed Floating-Point Values</b>	
xmmreg3 and merge with ymmreg2 to ymmreg1, imm8	C4: rxb0_3: 0 ymmreg2 101:18:11 ymmreg1 xmmreg3: imm
mem and merge with ymmreg2 to ymmreg1, imm8	C4: rxb0_3: 0 ymmreg2 101:18:mod ymmreg1 r/m: imm
<b>VPERMILPD – Permute Double-Precision Floating-Point Values</b>	
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
<b>VPERMILPS – Permute Single-Precision Floating-Point Values</b>	
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

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
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
<b>VPERM2F128 – Permute Floating-Point Values</b>	
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
<b>VTESTPD/VTESTPS – Packed Bit Test</b>	
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		OP	
5	11011	0	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) $\leftarrow$ ST(0) + 32-bit memory	11011 000 : mod 000 r/m
ST(0) $\leftarrow$ ST(0) + 64-bit memory	11011 100 : mod 000 r/m
ST(d) $\leftarrow$ ST(0) + ST(i)	11011 d00 : 11 000 ST(i)
FADDP - Add and Pop	
ST(0) $\leftarrow$ 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)
<b>FCOMP - Compare Real and Pop</b>	
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)
<b>FCOMPP - Compare Real and Pop Twice</b>	11011 110 : 11 011 001
<b>FCOMIP - Compare Real, Set EFLAGS, and Pop</b>	11011 111 : 11 110 ST(i)
<b>FCOS - Cosine of ST(0)</b>	11011 001 : 1111 1111
<b>FDECSTP - Decrement Stack-Top Pointer</b>	11011 001 : 1111 0110
<b>FDIV - Divide</b>	
ST(0) $\leftarrow$ ST(0) $\div$ 32-bit memory	11011 000 : mod 110 r/m
ST(0) $\leftarrow$ ST(0) $\div$ 64-bit memory	11011 100 : mod 110 r/m
ST(d) $\leftarrow$ ST(0) $\div$ ST(i)	11011 d00 : 1111 R ST(i)
<b>FDIVP - Divide and Pop</b>	
ST(0) $\leftarrow$ ST(0) $\div$ ST(i)	11011 110 : 1111 1 ST(i)
<b>FDIVR - Reverse Divide</b>	
ST(0) $\leftarrow$ 32-bit memory $\div$ ST(0)	11011 000 : mod 111 r/m
ST(0) $\leftarrow$ 64-bit memory $\div$ ST(0)	11011 100 : mod 111 r/m
ST(d) $\leftarrow$ ST(i) $\div$ ST(0)	11011 d00 : 1111 R ST(i)
<b>FDIVRP - Reverse Divide and Pop</b>	
ST(0) " ST(i) $\div$ ST(0)	11011 110 : 1111 0 ST(i)
<b>FFREE - Free ST(i) Register</b>	11011 101 : 1100 0 ST(i)
<b>FIADD - Add Integer</b>	
ST(0) $\leftarrow$ ST(0) + 16-bit memory	11011 110 : mod 000 r/m
ST(0) $\leftarrow$ ST(0) + 32-bit memory	11011 010 : mod 000 r/m
<b>FICOM - Compare Integer</b>	
16-bit memory	11011 110 : mod 010 r/m
32-bit memory	11011 010 : mod 010 r/m
<b>FICOMP - Compare Integer and Pop</b>	
16-bit memory	11011 110 : mod 011 r/m
32-bit memory	11011 010 : mod 011 r/m
<b>FIDIV - Divide</b>	
ST(0) $\leftarrow$ ST(0) $\div$ 16-bit memory	11011 110 : mod 110 r/m
ST(0) $\leftarrow$ ST(0) $\div$ 32-bit memory	11011 010 : mod 110 r/m
<b>FIDIVR - Reverse Divide</b>	
ST(0) $\leftarrow$ 16-bit memory $\div$ ST(0)	11011 110 : mod 111 r/m

**Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
$ST(0) \leftarrow 32\text{-bit memory} \div ST(0)$	11011 010 : mod 111 r/m
<b>FILD - Load Integer</b>	
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
<b>FIMUL - Multiply</b>	
$ST(0) \leftarrow ST(0) \times 16\text{-bit memory}$	11011 110 : mod 001 r/m
$ST(0) \leftarrow ST(0) \times 32\text{-bit memory}$	11011 010 : mod 001 r/m
<b>FINCSTP - Increment Stack Pointer</b>	11011 001 : 1111 0111
<b>FINIT - Initialize Floating-Point Unit</b>	
<b>FIST - Store Integer</b>	
16-bit memory	11011 111 : mod 010 r/m
32-bit memory	11011 011 : mod 010 r/m
<b>FISTP - Store Integer and Pop</b>	
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
<b>FISUB - Subtract</b>	
$ST(0) \leftarrow ST(0) - 16\text{-bit memory}$	11011 110 : mod 100 r/m
$ST(0) \leftarrow ST(0) - 32\text{-bit memory}$	11011 010 : mod 100 r/m
<b>FISUBR - Reverse Subtract</b>	
$ST(0) \leftarrow 16\text{-bit memory} - ST(0)$	11011 110 : mod 101 r/m
$ST(0) \leftarrow 32\text{-bit memory} - ST(0)$	11011 010 : mod 101 r/m
<b>FLD - Load Real</b>	
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)
<b>FLD1 - Load +1.0 into ST(0)</b>	11011 001 : 1110 1000
<b>FLDCW - Load Control Word</b>	11011 001 : mod 101 r/m
<b>FLDENV - Load FPU Environment</b>	11011 001 : mod 100 r/m
<b>FLDL2E - Load <math>\log_2(\epsilon)</math> into ST(0)</b>	11011 001 : 1110 1010
<b>FLDL2T - Load <math>\log_2(10)</math> into ST(0)</b>	11011 001 : 1110 1001
<b>FLDLG2 - Load <math>\log_{10}(2)</math> into ST(0)</b>	11011 001 : 1110 1100
<b>FLDLN2 - Load <math>\log_e(2)</math> into ST(0)</b>	11011 001 : 1110 1101
<b>FLDPI - Load <math>\pi</math> into ST(0)</b>	11011 001 : 1110 1011
<b>FLDZ - Load +0.0 into ST(0)</b>	11011 001 : 1110 1110
<b>FMUL - Multiply</b>	

**Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
ST(0) $\leftarrow$ ST(0) $\times$ 32-bit memory	11011 000 : mod 001 r/m
ST(0) $\leftarrow$ ST(0) $\times$ 64-bit memory	11011 100 : mod 001 r/m
ST(d) $\leftarrow$ ST(0) $\times$ ST(i)	11011 d00 : 1100 1 ST(i)
<b>FMULP - Multiply</b>	
ST(i) $\leftarrow$ ST(0) $\times$ ST(i)	11011 110 : 1100 1 ST(i)
<b>FNOP - No Operation</b>	11011 001 : 1101 0000
<b>FPATAN - Partial Arctangent</b>	11011 001 : 1111 0011
<b>FPREM - Partial Remainder</b>	11011 001 : 1111 1000
<b>FPREM1 - Partial Remainder (IEEE)</b>	11011 001 : 1111 0101
<b>FPTAN - Partial Tangent</b>	11011 001 : 1111 0010
<b>FRNDINT - Round to Integer</b>	11011 001 : 1111 1100
<b>FRSTOR - Restore FPU State</b>	11011 101 : mod 100 r/m
<b>FSAVE - Store FPU State</b>	11011 101 : mod 110 r/m
<b>FSCALE - Scale</b>	11011 001 : 1111 1101
<b>FSIN - Sine</b>	11011 001 : 1111 1110
<b>FSINCOS - Sine and Cosine</b>	11011 001 : 1111 1011
<b>FSQRT - Square Root</b>	11011 001 : 1111 1010
<b>FST - Store Real</b>	
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)
<b>FSTCW - Store Control Word</b>	11011 001 : mod 111 r/m
<b>FSTENV - Store FPU Environment</b>	11011 001 : mod 110 r/m
<b>FSTP - Store Real and Pop</b>	
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)
<b>FSTSW - Store Status Word into AX</b>	11011 111 : 1110 0000
<b>FSTSW - Store Status Word into Memory</b>	11011 101 : mod 111 r/m
<b>FSUB - Subtract</b>	
ST(0) $\leftarrow$ ST(0) - 32-bit memory	11011 000 : mod 100 r/m
ST(0) $\leftarrow$ ST(0) - 64-bit memory	11011 100 : mod 100 r/m
ST(d) $\leftarrow$ ST(0) - ST(i)	11011 d00 : 1110 R ST(i)
<b>FSUBP - Subtract and Pop</b>	
ST(0) $\leftarrow$ ST(0) - ST(i)	11011 110 : 1110 1 ST(i)
<b>FSUBR - Reverse Subtract</b>	
ST(0) $\leftarrow$ 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)
<b>FSUBRP – Reverse Subtract and Pop</b>	
ST(i) ← ST(i) – ST(0)	11011 110 : 1110 0 ST(i)
<b>FTST – Test</b>	11011 001 : 1110 0100
<b>FUCOM – Unordered Compare Real</b>	11011 101 : 1110 0 ST(i)
<b>FUCOMP – Unordered Compare Real and Pop</b>	11011 101 : 1110 1 ST(i)
<b>FUCOMPP – Unordered Compare Real and Pop Twice</b>	11011 010 : 1110 1001
<b>FUCOMI – Unorderd Compare Real and Set EFLAGS</b>	11011 011 : 11 101 ST(i)
<b>FUCOMIP – Unorderd Compare Real, Set EFLAGS, and Pop</b>	11011 111 : 11 101 ST(i)
<b>FXAM – Examine</b>	11011 001 : 1110 0101
<b>FXCH – Exchange ST(0) and ST(i)</b>	11011 001 : 1100 1 ST(i)
<b>FXTRACT – Extract Exponent and Significand</b>	11011 001 : 1111 0100
<b>FYL2X – ST(1) × log<sub>2</sub>(ST(0))</b>	11011 001 : 1111 0001
<b>FYL2XP1 – ST(1) × log<sub>2</sub>(ST(0) + 1.0)</b>	11011 001 : 1111 1001
<b>FWAIT – Wait until FPU Ready</b>	1001 1011

## B.18 VMX INSTRUCTIONS

Table B-40 describes virtual-machine extensions (VMX).

**Table B-40. Encodings for VMX Instructions**

Instruction and Format	Encoding
<b>INVEPT—Invalidate Cached EPT Mappings</b>	
Descriptor m128 according to reg	01100110 00001111 00111000 10000000: mod reg r/m
<b>INVVPID—Invalidate Cached VPID Mappings</b>	
Descriptor m128 according to reg	01100110 00001111 00111000 10000001: mod reg r/m
<b>VMCALL—Call to VM Monitor</b>	
Call VMM: causes VM exit.	00001111 00000001 11000001
<b>VMCLEAR—Clear Virtual-Machine Control Structure</b>	
mem32:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
mem64:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
<b>VMFUNC—Invoke VM Function</b>	
Invoke VM function specified in EAX	00001111 00000001 11010100
<b>VMLAUNCH—Launch Virtual Machine</b>	
Launch VM managed by Current_VMCS	00001111 00000001 11000010
<b>VMRESUME—Resume Virtual Machine</b>	
Resume VM managed by Current_VMCS	00001111 00000001 11000011
<b>VMPTRLD—Load Pointer to Virtual-Machine Control Structure</b>	

**Table B-40. Encodings for VMX Instructions**

Instruction and Format	Encoding
mem32 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m
mem64 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m
<b>VMPTRST—Store Pointer to Virtual-Machine Control Structure</b>	
Current_VMCS_ptr to mem32	00001111 11000111: mod 111 r/m
Current_VMCS_ptr to mem64	00001111 11000111: mod 111 r/m
<b>VMREAD—Read Field from Virtual-Machine Control Structure</b>	
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
<b>VMWRITE—Write Field to Virtual-Machine Control Structure</b>	
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
<b>VMXOFF—Leave VMX Operation</b>	
Leave VMX.	00001111 00000001 11000100
<b>VMXON—Enter VMX Operation</b>	
Enter VMX.	11110011 000011111 11000111: mod 110 r/m

## B.19 SMX INSTRUCTIONS

Table B-38 describes Safer Mode extensions (VMX). **GETSEC leaf functions are selected by a valid value in EAX on input.**

**Table B-41. Encodings for SMX Instructions**

Instruction and Format	Encoding
<b>GETSEC—GETSEC leaf functions are selected by the value in EAX on input</b>	
<i>GETSEC[CAPABILITIES].</i>	00001111 00110111 (EAX= 0)
<i>GETSEC[ENTERACCS].</i>	00001111 00110111 (EAX= 2)
<i>GETSEC[EXITAC].</i>	00001111 00110111 (EAX= 3)
<i>GETSEC[SENTER].</i>	00001111 00110111 (EAX= 4)
<i>GETSEC[SEXIT].</i>	00001111 00110111 (EAX= 5)
<i>GETSEC[PARAMETERS].</i>	00001111 00110111 (EAX= 6)
<i>GETSEC[SMCTRL].</i>	00001111 00110111 (EAX= 7)
<i>GETSEC[WAKEUP].</i>	00001111 00110111 (EAX= 8)

## 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:

- |                                |   |
|--------------------------------|---|
| <code>&lt;intrin_op&gt;</code> | Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction   |
| <code>&lt;suffix&gt;</code>    | 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.

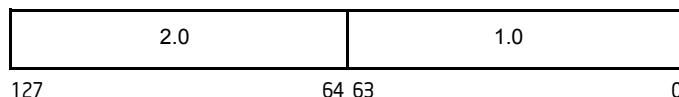
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
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 in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A”, or Chapter 4, “Instruction Set Reference, N-Z” in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B”.

**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)
BLENDPSP	__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask)
BLENDPVD	__m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3)
BLENDPVS	__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)
CLFLUSH	void _mm_clflush(void const *p)
CMPPD	__m128d _mm_cmpeq_pd(__m128d a, __m128d b) __m128d _mm_cmplt_pd(__m128d a, __m128d b)

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	<code>__m128d_mm_cmple_pd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmpgt_pd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmpte_pd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmpeq_pd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmplt_pd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmptge_pd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmprord_pd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmputord_pd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmplne_pd(__m128d a, __m128d b)</code>
CMPPS	<code>__m128_mm_cmpeq_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmplt_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmple_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmptgt_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmpte_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmpeq_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmplt_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmptge_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmprord_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmputord_ps(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmplne_ps(__m128 a, __m128 b)</code>
CMPSD	<code>__m128d_mm_cmpeq_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmplt_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmple_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmptgt_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmpte_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmpeq_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmplt_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmptge_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmprord_sd(__m128d a, __m128d b)</code>
	<code>__m128d_mm_cmputord_sd(__m128d a, __m128d b)</code>
CMPSS	<code>__m128_mm_cmpeq_ss(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmplt_ss(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmple_ss(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmptgt_ss(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmpte_ss(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmpeq_ss(__m128 a, __m128 b)</code>
	<code>__m128_mm_cmplt_ss(__m128 a, __m128 b)</code>

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	<code>_m128_mm_cmple_ss(_m128 a, _m128 b)</code>
	<code>_m128_mm_cmpngt_ss(_m128 a, _m128 b)</code>
	<code>_m128_mm_cmprnge_ss(_m128 a, _m128 b)</code>
	<code>_m128_mm_cmppord_ss(_m128 a, _m128 b)</code>
	<code>_m128_mm_cmppunord_ss(_m128 a, _m128 b)</code>
COMISD	<code>int_mm_comieq_sd(_m128d a, _m128d b)</code> <code>int_mm_comilt_sd(_m128d a, _m128d b)</code> <code>int_mm_comile_sd(_m128d a, _m128d b)</code> <code>int_mm_comigt_sd(_m128d a, _m128d b)</code> <code>int_mm_comige_sd(_m128d a, _m128d b)</code> <code>int_mm_comineq_sd(_m128d a, _m128d b)</code>
COMISS	<code>int_mm_comieq_ss(_m128 a, _m128 b)</code> <code>int_mm_comilt_ss(_m128 a, _m128 b)</code> <code>int_mm_comile_ss(_m128 a, _m128 b)</code> <code>int_mm_comigt_ss(_m128 a, _m128 b)</code> <code>int_mm_comige_ss(_m128 a, _m128 b)</code> <code>int_mm_comineq_ss(_m128 a, _m128 b)</code>
CRC32	<code>unsigned int_mm_crc32_u8(unsigned int crc, unsigned char data)</code> <code>unsigned int_mm_crc32_u16(unsigned int crc, unsigned short data)</code> <code>unsigned int_mm_crc32_u32(unsigned int crc, unsigned int data)</code> <code>unsigned __int64 _mm_crc32_u64(unsigned __int64 crc, unsigned __int64 data)</code>
CVTDQ2PD	<code>_m128d_mm_cvtepi32_pd(_m128i a)</code>
CVTDQ2PS	<code>_m128_mm_cvtepi32_ps(_m128i a)</code>
CVTPD2DQ	<code>_m128i_mm_cvtpd_epi32(_m128d a)</code>
CVTPD2PI	<code>_m64_mm_cvtpd_pi32(_m128d a)</code>
CVTPD2PS	<code>_m128_mm_cvtpd_ps(_m128d a)</code>
CVTPI2PD	<code>_m128d_mm_cvtpi32_pd(_m64 a)</code>
CVTPI2PS	<code>_m128_mm_cvt_pi2ps(_m128 a, _m64 b)</code> <code>_m128_mm_cvtpi32_ps(_m128 a, _m64 b)</code>
CVTPS2DQ	<code>_m128i_mm_cvtps_epi32(_m128 a)</code>
CVTPS2PD	<code>_m128d_mm_cvtps_pd(_m128 a)</code>
CVTPS2PI	<code>_m64_mm_cvt_ps2pi(_m128 a)</code> <code>_m64_mm_cvtps_pi32(_m128 a)</code>
CVTSD2SI	<code>int_mm_cvtsd_si32(_m128d a)</code>
CVTSD2SS	<code>_m128_mm_cvtsd_ss(_m128 a, _m128d b)</code>
CVTSI2SD	<code>_m128d_mm_cvtsi32_sd(_m128d a, int b)</code>
CVTSI2SS	<code>_m128_mm_cvt_si2ss(_m128 a, int b)</code> <code>_m128_mm_cvtsi32_ss(_m128 a, int b)</code> <code>_m128_mm_cvtsi64_ss(_m128 a, __int64 b)</code>
CVTSS2SD	<code>_m128d_mm_cvtss_sd(_m128d a, _m128 b)</code>
CVTSS2SI	<code>int_mm_cvt_ss2si(_m128 a)</code> <code>int_mm_cvtss_si32(_m128 a)</code>
CVTPPD2DQ	<code>_m128i_mm_cvtpd_epi32(_m128d a)</code>

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
CVTPD2PI	<code>__m64_mm_cvtpd_pi32(__m128d a)</code>
CVTPS2DQ	<code>__m128i_mm_cvtps_epi32(__m128 a)</code>
CVTPS2PI	<code>__m64_mm_cvtt_ps2pi(__m128 a)</code> <code>__m64_mm_cvtps_pi32(__m128 a)</code>
CVTSD2SI	<code>int_mm_cvttsd_si32(__m128d a)</code>
CVTSS2SI	<code>int_mm_cvtt_ss2si(__m128 a)</code> <code>int_mm_cvtss_si32(__m128 a)</code> <code>__m64_mm_cvtsi32_si64(int i)</code> <code>int_mm_cvtsi64_si32(__m64 m)</code>
DIVPD	<code>__m128d_mm_div_pd(__m128d a, __m128d b)</code>
DIVPS	<code>__m128_mm_div_ps(__m128 a, __m128 b)</code>
DIVSD	<code>__m128d_mm_div_sd(__m128d a, __m128d b)</code>
DIVSS	<code>__m128_mm_div_ss(__m128 a, __m128 b)</code>
DPPD	<code>__m128d_mm_dp_pd(__m128d a, __m128d b, const int mask)</code>
DPPS	<code>__m128_mm_dp_ps(__m128 a, __m128 b, const int mask)</code>
EMMS	<code>void_mm_empty()</code>
EXTRACTPS	<code>int_mm_extract_ps(__m128 src, const int ndx)</code>
HADDPD	<code>__m128d_mm_hadd_pd(__m128d a, __m128d b)</code>
HADDPS	<code>__m128_mm_hadd_ps(__m128 a, __m128 b)</code>
HSUBPD	<code>__m128d_mm_hsub_pd(__m128d a, __m128d b)</code>
HSUBPS	<code>__m128_mm_hsub_ps(__m128 a, __m128 b)</code>
INSERTPS	<code>__m128_mm_insert_ps(__m128 dst, __m128 src, const int ndx)</code>
LDDQU	<code>__m128i_mm_lddqu_si128(__m128i const *p)</code>
LDMXCSR	<code>__mm_setcsr(unsigned int i)</code>
LFENCE	<code>void_mm_lfence(void)</code>
MASKMOVDQU	<code>void_mm_maskmoveu_si128(__m128i d, __m128i n, char *p)</code>
MASKMOVQ	<code>void_mm_maskmove_si64(__m64 d, __m64 n, char *p)</code>
MAXPD	<code>__m128d_mm_max_pd(__m128d a, __m128d b)</code>
MAXPS	<code>__m128_mm_max_ps(__m128 a, __m128 b)</code>
MAXSD	<code>__m128d_mm_max_sd(__m128d a, __m128d b)</code>
MAXSS	<code>__m128_mm_max_ss(__m128 a, __m128 b)</code>
MFENCE	<code>void_mm_mfence(void)</code>
MINPD	<code>__m128d_mm_min_pd(__m128d a, __m128d b)</code>
MINPS	<code>__m128_mm_min_ps(__m128 a, __m128 b)</code>
MINSD	<code>__m128d_mm_min_sd(__m128d a, __m128d b)</code>
MINSS	<code>__m128_mm_min_ss(__m128 a, __m128 b)</code>
MONITOR	<code>void_mm_monitor(void const *p, unsigned extensions, unsigned hints)</code>
MOVAPD	<code>__m128d_mm_load_pd(double * p)</code>
	<code>void_mm_store_pd(double *p, __m128d a)</code>
MOVAPS	<code>__m128_mm_load_ps(float * p)</code>
	<code>void_mm_store_ps(float *p, __m128 a)</code>
MOVD	<code>__m128i_mm_cvtsi32_si128(int a)</code>

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	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_loadaddup_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_storl_pd(double * p, _m128d a)
MOVLPS	_m128 _mm_loadl_pi(_m128 a, _m64 * p)
	void _mm_storl_pi(_m64 * p, _m128 a)
MOVHLPS	_m128 _mm_movehl_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_storl_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)
	void _mm_store_ss(float * p, _m128 a)
	_m128 _mm_move_ss(_m128 a, _m128 b)
MOVUPD	_m128d _mm_loadu_pd(double * p)

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	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)
PADDUSW	__m128i _mm_adds_epu16(__m128i m1, __m128i m2)
PADDUSW	__m64 _mm_adds_pu16(__m64 m1, __m64 m2)

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
PALIGNR	<code>_m64_mm_alignr_pi8(_m64 a, _m64 b, int n)</code> <code>_m128i_mm_alignr_epi8(_m128i a, _m128i b, int n)</code>
PAND	<code>_m128i_mm_and_si128(_m128i m1, _m128i m2)</code>
PAND	<code>_m64_mm_and_si64(_m64 m1, _m64 m2)</code>
PANDN	<code>_m128i_mm_andnot_si128(_m128i m1, _m128i m2)</code>
PANDN	<code>_m64_mm_andnot_si64(_m64 m1, _m64 m2)</code>
PAUSE	<code>void_mm_pause(void)</code>
PAVGB	<code>_m128i_mm_avg_epu8(_m128i a, _m128i b)</code>
PAVGB	<code>_m64_mm_avg_pu8(_m64 a, _m64 b)</code>
PAVGW	<code>_m128i_mm_avg_epu16(_m128i a, _m128i b)</code>
PAVGW	<code>_m64_mm_avg_pu16(_m64 a, _m64 b)</code>
PBLENDVB	<code>_m128i_mm_blendv_epi(_m128i v1, _m128i v2, _m128i mask)</code>
PBLENDW	<code>_m128i_mm_blend_epi16(_m128i v1, _m128i v2, const int mask)</code>
PCLMULQDQ	<code>_m128i_mm_clmulepi64_si128(_m128i, _m128i, const int)</code>
PCMPEQB	<code>_m128i_mm_cmpeq_epi8(_m128i m1, _m128i m2)</code>
PCMPEQB	<code>_m64_mm_cmpeq_pi8(_m64 m1, _m64 m2)</code>
PCMPEQQ	<code>_m128i_mm_cmpeq_epi64(_m128i a, _m128i b)</code>
PCMPEQW	<code>_m128i_mm_cmpeq_epi16(_m128i m1, _m128i m2)</code>
PCMPEQW	<code>_m64_mm_cmpeq_pi16(_m64 m1, _m64 m2)</code>
PCMPEQD	<code>_m128i_mm_cmpeq_epi32(_m128i m1, _m128i m2)</code>
PCMPEQD	<code>_m64_mm_cmpeq_pi32(_m64 m1, _m64 m2)</code>
PCMPESTRI	<code>int_mm_cmpestri(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestr(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestrc(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestro(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestrs(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestrz(_m128i a, int la, _m128i b, int lb, const int mode)</code>
PCMPESTRM	<code>_m128i_mm_cmpestrm(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestra(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestrc(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestro(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestrs(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestrz(_m128i a, int la, _m128i b, int lb, const int mode)</code>
PCMPGTB	<code>_m128i_mm_cmpgt_epi8(_m128i m1, _m128i m2)</code>
PCMPGTB	<code>_m64_mm_cmpgt_pi8(_m64 m1, _m64 m2)</code>
PCMPGTw	<code>_m128i_mm_cmpgt_epi16(_m128i m1, _m128i m2)</code>
PCMPGTw	<code>_m64_mm_cmpgt_pi16(_m64 m1, _m64 m2)</code>
PCMPGTD	<code>_m128i_mm_cmpgt_epi32(_m128i m1, _m128i m2)</code>
PCMPGTD	<code>_m64_mm_cmpgt_pi32(_m64 m1, _m64 m2)</code>
PCMPISTRI	<code>_m128i_mm_cmpestrm(_m128i a, int la, _m128i b, int lb, const int mode)</code> <code>int_mm_cmpestra(_m128i a, int la, _m128i b, int lb, const int mode)</code>

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	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_cmppgt_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)
PMAXSB	_m128i _mm_max_epi8( _m128i a, _m128i b)
PMAXSD	_m128i _mm_max_epi32( _m128i a, _m128i b)

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
PMAXSW	<code>_m128i_mm_max_epi16(_m128i a, _m128i b)</code>
PMAXSW	<code>_m64_mm_max_pi16(_m64 a, _m64 b)</code>
PMAXUB	<code>_m128i_mm_max_epu8(_m128i a, _m128i b)</code>
PMAXUB	<code>_m64_mm_max_pu8(_m64 a, _m64 b)</code>
PMAXUD	<code>_m128i_mm_max_epu32(_m128i a, _m128i b)</code>
PMAXUW	<code>_m128i_mm_max_epu16(_m128i a, _m128i b)</code>
PMINSB	<code>_m128i_mm_min_epi8(_m128i a, _m128i b)</code>
PMINSD	<code>_m128i_mm_min_epi32(_m128i a, _m128i b)</code>
PMINSW	<code>_m128i_mm_min_epi16(_m128i a, _m128i b)</code>
PMINSW	<code>_m64_mm_min_pi16(_m64 a, _m64 b)</code>
PMINUB	<code>_m128i_mm_min_epu8(_m128i a, _m128i b)</code>
PMINUB	<code>_m64_mm_min_pu8(_m64 a, _m64 b)</code>
PMINUD	<code>_m128i_mm_min_epu32(_m128i a, _m128i b)</code>
PMINUW	<code>_m128i_mm_min_epu16(_m128i a, _m128i b)</code>
PMOVMSKB	<code>int_mm_movemask_epi8(_m128i a)</code>
PMOVMSKB	<code>int_mm_movemask_pi8(_m64 a)</code>
PMOVSXBW	<code>_m128i_mm_cvtepi8_epi16(_m128i a)</code>
PMOVSXBD	<code>_m128i_mm_cvtepi8_epi32(_m128i a)</code>
PMOVSXBQ	<code>_m128i_mm_cvtepi8_epi64(_m128i a)</code>
PMOVSXWD	<code>_m128i_mm_cvtepi16_epi32(_m128i a)</code>
PMOVSXWQ	<code>_m128i_mm_cvtepi16_epi64(_m128i a)</code>
PMOVSXDQ	<code>_m128i_mm_cvtepi32_epi64(_m128i a)</code>
PMOVZXBW	<code>_m128i_mm_cvtepu8_epi16(_m128i a)</code>
PMOVZXBD	<code>_m128i_mm_cvtepu8_epi32(_m128i a)</code>
PMOVZXBQ	<code>_m128i_mm_cvtepu8_epi64(_m128i a)</code>
PMOVZXWD	<code>_m128i_mm_cvtepu16_epi32(_m128i a)</code>
PMOVZXWQ	<code>_m128i_mm_cvtepu16_epi64(_m128i a)</code>
PMOVZXDQ	<code>_m128i_mm_cvtepu32_epi64(_m128i a)</code>
PMULDQ	<code>_m128i_mm_mul_epi32(_m128i a, _m128i b)</code>
PMULHRSW	<code>_m64_mm_mulhrs_pi16(_m64 a, _m64 b)</code>
	<code>_m128i_mm_mulhrs_epi16(_m128i a, _m128i b)</code>
PMULHUW	<code>_m128i_mm_mulhi_epu16(_m128i a, _m128i b)</code>
PMULHUW	<code>_m64_mm_mulhi_pu16(_m64 a, _m64 b)</code>
PMULHW	<code>_m128i_mm_mulhi_epi16(_m128i m1, _m128i m2)</code>
PMULHW	<code>_m64_mm_mulhi_pi16(_m64 m1, _m64 m2)</code>
PMULLUD	<code>_m128i_mm_mullo_epi32(_m128i a, _m128i b)</code>
PMULLW	<code>_m128i_mm_mullo_epi16(_m128i m1, _m128i m2)</code>
PMULLW	<code>_m64_mm_mullo_pi16(_m64 m1, _m64 m2)</code>
PMULUDQ	<code>_m64_mm_mul_su32(_m64 m1, _m64 m2)</code>
	<code>_m128i_mm_mul_epu32(_m128i m1, _m128i m2)</code>

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
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_slli_epi16(_m128i m, _m128i count)
PSLLW	_m128i_mm_slli_epi16(_m128i m, int count)
PSLLW	_m64_mm_slli_pi16(_m64 m, _m64 count)
	_m64_mm_slli_pi16(_m64 m, int count)
PSLLD	_m128i_mm_slli_epi32(_m128i m, int count)
	_m128i_mm_slli_epi32(_m128i m, _m128i count)
PSLLD	_m64_mm_slli_pi32(_m64 m, int count)
	_m64_mm_slli_pi32(_m64 m, _m64 count)
PSLLQ	_m64_mm_slli_si64(_m64 m, _m64 count)
	_m64_mm_slli_si64(_m64 m, int count)
PSLLQ	_m128i_mm_slli_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)
	_m64_mm_srai_pi32 (_m64 m, int count)
PSRLW	_m128i_mm_srl_epi16 (_m128i m, _m128i count)

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	<code>_m128i_mm_srli_epi16(_m128i m, int count)</code>
	<code>_m64_mm_srl_pi16(_m64 m, _m64 count)</code>
	<code>_m64_mm_srli_pi16(_m64 m, int count)</code>
PSRLD	<code>_m128i_mm_srl_epi32(_m128i m, _m128i count)</code>
	<code>_m128i_mm_srli_epi32(_m128i m, int count)</code>
PSRLD	<code>_m64_mm_srl_pi32(_m64 m, _m64 count)</code>
	<code>_m64_mm_srli_pi32(_m64 m, int count)</code>
PSRLQ	<code>_m128i_mm_srl_epi64(_m128i m, _m128i count)</code>
	<code>_m128i_mm_srli_epi64(_m128i m, int count)</code>
PSRLQ	<code>_m64_mm_srl_si64(_m64 m, _m64 count)</code>
	<code>_m64_mm_srli_si64(_m64 m, int count)</code>
PSRLDQ	<code>_m128i_mm_srli_si128(_m128i m, int imm)</code>
PSUBB	<code>_m128i_mm_sub_epi8(_m128i m1, _m128i m2)</code>
PSUBB	<code>_m64_mm_sub_pi8(_m64 m1, _m64 m2)</code>
PSUBW	<code>_m128i_mm_sub_epi16(_m128i m1, _m128i m2)</code>
PSUBW	<code>_m64_mm_sub_pi16(_m64 m1, _m64 m2)</code>
PSUBD	<code>_m128i_mm_sub_epi32(_m128i m1, _m128i m2)</code>
PSUBD	<code>_m64_mm_sub_pi32(_m64 m1, _m64 m2)</code>
PSUBQ	<code>_m128i_mm_sub_epi64(_m128i m1, _m128i m2)</code>
PSUBQ	<code>_m64_mm_sub_si64(_m64 m1, _m64 m2)</code>
PSUBSB	<code>_m128i_mm_subs_epi8(_m128i m1, _m128i m2)</code>
PSUBSB	<code>_m64_mm_subs_pi8(_m64 m1, _m64 m2)</code>
PSUBSW	<code>_m128i_mm_subs_epi16(_m128i m1, _m128i m2)</code>
PSUBSW	<code>_m64_mm_subs_pi16(_m64 m1, _m64 m2)</code>
PSUBUSB	<code>_m128i_mm_subs_epu8(_m128i m1, _m128i m2)</code>
PSUBUSB	<code>_m64_mm_subs_pu8(_m64 m1, _m64 m2)</code>
PSUBUSW	<code>_m128i_mm_subs_epu16(_m128i m1, _m128i m2)</code>
PSUBUSW	<code>_m64_mm_subs_pu16(_m64 m1, _m64 m2)</code>
PTEST	<code>int_mm_testz_si128(_m128i s1, _m128i s2)</code>
	<code>int_mm_testc_si128(_m128i s1, _m128i s2)</code>
	<code>int_mm_testnc_si128(_m128i s1, _m128i s2)</code>
PUNPCKHBW	<code>_m64_mm_unpackhi_pi8(_m64 m1, _m64 m2)</code>
PUNPCKHBW	<code>_m128i_mm_unpackhi_epi8(_m128i m1, _m128i m2)</code>
PUNPCKHWD	<code>_m64_mm_unpackhi_pi16(_m64 m1, _m64 m2)</code>
PUNPCKHWD	<code>_m128i_mm_unpackhi_epi16(_m128i m1, _m128i m2)</code>
PUNPCKHDQ	<code>_m64_mm_unpackhi_pi32(_m64 m1, _m64 m2)</code>
PUNPCKHDQ	<code>_m128i_mm_unpackhi_epi32(_m128i m1, _m128i m2)</code>
PUNPCKHQDQ	<code>_m128i_mm_unpackhi_epi64(_m128i m1, _m128i m2)</code>
PUNPCKLBW	<code>_m64_mm_unpacklo_pi8(_m64 m1, _m64 m2)</code>
PUNPCKLBW	<code>_m128i_mm_unpacklo_epi8(_m128i m1, _m128i m2)</code>
PUNPCKLWD	<code>_m64_mm_unpacklo_pi16(_m64 m1, _m64 m2)</code>

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
PUNPCKLWD	<code>_m128i_mm_unpacklo_epi16(_m128i m1, _m128i m2)</code>
PUNPCKLDQ	<code>_m64_mm_unpacklo_pi32(_m64 m1, _m64 m2)</code>
PUNPCKLDQ	<code>_m128i_mm_unpacklo_epi32(_m128i m1, _m128i m2)</code>
PUNPCKLQDQ	<code>_m128i_mm_unpacklo_epi64(_m128i m1, _m128i m2)</code>
PXOR	<code>_m64_mm_xor_si64(_m64 m1, _m64 m2)</code>
PXOR	<code>_m128i_mm_xor_si128(_m128i m1, _m128i m2)</code>
RCPSS	<code>_m128_mm_rcp_ps(_m128 a)</code>
RCPSS	<code>_m128_mm_rcp_ss(_m128 a)</code>
ROUNDPD	<code>_m128_mm_round_pd(_m128d s1, int iRoundMode)</code>
	<code>_m128_mm_floor_pd(_m128d s1)</code>
	<code>_m128_mm_ceil_pd(_m128d s1)</code>
ROUNDPS	<code>_m128_mm_round_ps(_m128 s1, int iRoundMode)</code>
	<code>_m128_mm_floor_ps(_m128 s1)</code>
	<code>_m128_mm_ceil_ps(_m128 s1)</code>
ROUNDSD	<code>_m128d_mm_round_sd(_m128d dst, _m128d s1, int iRoundMode)</code>
	<code>_m128d_mm_floor_sd(_m128d dst, _m128d s1)</code>
	<code>_m128d_mm_ceil_sd(_m128d dst, _m128d s1)</code>
ROUNDSS	<code>_m128_mm_round_ss(_m128 dst, _m128 s1, int iRoundMode)</code>
	<code>_m128_mm_floor_ss(_m128 dst, _m128 s1)</code>
	<code>_m128_mm_ceil_ss(_m128 dst, _m128 s1)</code>
RSQRTPS	<code>_m128_mm_rsqrt_ps(_m128 a)</code>
RSQRTSS	<code>_m128_mm_rsqrt_ss(_m128 a)</code>
SFENCE	<code>void_mm_sfence(void)</code>
SHUFPD	<code>_m128d_mm_shuffle_pd(_m128d a, _m128d b, unsigned int imm8)</code>
SHUFPS	<code>_m128_mm_shuffle_ps(_m128 a, _m128 b, unsigned int imm8)</code>
SQRTPD	<code>_m128d_mm_sqrt_pd(_m128d a)</code>
SQRTPS	<code>_m128_mm_sqrt_ps(_m128 a)</code>
SQRTSD	<code>_m128d_mm_sqrt_sd(_m128d a)</code>
SQRTSS	<code>_m128_mm_sqrt_ss(_m128 a)</code>
STMXCSR	<code>_mm_getcsr(void)</code>
SUBPD	<code>_m128d_mm_sub_pd(_m128d a, _m128d b)</code>
SUBPS	<code>_m128_mm_sub_ps(_m128 a, _m128 b)</code>
SUBSD	<code>_m128d_mm_sub_sd(_m128d a, _m128d b)</code>
SUBSS	<code>_m128_mm_sub_ss(_m128 a, _m128 b)</code>
UCOMISD	<code>int_mm_ucomieq_sd(_m128d a, _m128d b)</code>
	<code>int_mm_ucomilt_sd(_m128d a, _m128d b)</code>
	<code>int_mm_ucomile_sd(_m128d a, _m128d b)</code>
	<code>int_mm_ucomigt_sd(_m128d a, _m128d b)</code>
	<code>int_mm_ucomige_sd(_m128d a, _m128d b)</code>
	<code>int_mm_ucomineq_sd(_m128d a, _m128d b)</code>
UCOMISS	<code>int_mm_ucomieq_ss(_m128 a, _m128 b)</code>

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	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)
(composite)	_m128d_mm_setzero_pd(void)
(composite)	_m128_mm_setzero_ps(void)

**Table C-2. Composite Intrinsics (Contd.)**

Mnemonic	Intrinsic
MOVSD + shuffle	<code>__m128d _mm_load_pd(double * p)</code> <code>__m128d _mm_load1_pd(double *p)</code>
MOVSS + shuffle	<code>__m128 _mm_load_ps1(float * p)</code> <code>__m128 _mm_load1_ps(float *p)</code>
MOVAPD + shuffle	<code>__m128d _mm_loadr_pd(double * p)</code>
MOVAPS + shuffle	<code>__m128 _mm_loadr_ps(float * p)</code>
MOVSD + shuffle	<code>void _mm_store1_pd(double *p, __m128d a)</code>
MOVSS + shuffle	<code>void _mm_store_ps1(float * p, __m128 a)</code> <code>void _mm_store1_ps(float *p, __m128 a)</code>
MOVAPD + shuffle	<code>_mm_storer_pd(double * p, __m128d a)</code>
MOVAPS + shuffle	<code>_mm_storer_ps(float * p, __m128 a)</code>



**Numerics**

0000 B-41

64-bit mode

- control and debug registers 2-11
- default operand size 2-11
- direct memory-offset MOVs 2-10
- general purpose encodings B-18
- immediates 2-10
- introduction 2-6
- machine instructions B-1
- reg (reg) field B-3
- REX prefixes 2-7, B-2
- RIP-relative addressing 2-11
- SIMD encodings B-37
- special instruction encodings B-64
- summary table notation 3-7

**A**

AAA instruction 3-16, 3-18

AAD instruction 3-18

AAM instruction 3-20

AAS instruction 3-22

ADC instruction 3-24, 3-468

ADD instruction 3-16, 3-27, 3-229, 3-468

ADDPD instruction 3-29

ADDPS instruction 3-31

Addressing methods

- RIP-relative 2-11

Addressing, segments 1-5

ADDSD instruction 3-33

ADDSS instruction 3-34

ADDSUBPD instruction 3-35

ADDSUBPS instruction 3-37

AESDEC/AESDECLAST - Perform One Round of an AES Decryption

Flow 3-46

AESIMC - Perform the AES InvMixColumn Transformation 3-42

AESKEYGENASSIST - AES Round Key Generation Assist 3-49

AND instruction 3-51, 3-468

ANDNPD instruction 3-58

ANDNPS instruction 3-60

ANDPD instruction 3-53

ANDPS instruction 3-56

Arctangent, x87 FPU operation 3-313

ARPL instruction 3-62

authenticated code execution mode 5-3

**B**

Base (operand addressing) 2-3

BCD integers

- packed 3-229, 3-231, 3-263, 3-265
- unpacked 3-16, 3-18, 3-20, 3-22

BEXTR - Bit Field Extract 3-66

Binary numbers 1-5

Bit order 1-3

BLSMSK - Get Mask Up to Lowest Set Bit 3-76

bootstrap processor 5-15, 5-20, 5-28, 5-29

BOUND instruction 3-78, 4-424

BOUND range exceeded exception (#BR) 3-78, 4-424

Branch hints 2-2

Brand information 3-179

- processor brand index 3-182

processor brand string 3-180

BSF instruction 3-80

BSR instruction 3-82

BSWAP instruction 3-84

BT instruction 3-85

BTC instruction 3-87, 3-468

BTR instruction 3-89, 3-468

BTS instruction 3-91, 3-468

Byte order 1-3

**C**

C/C++ compiler intrinsics

- compiler functional equivalents C-1

composite C-14

description of 3-11

lists of C-1

simple C-2

Cache and TLB information 3-174

Cache Inclusiveness 3-159

Caches, invalidating (flushing) 3-416, 4-547

CALL instruction 3-95

GETSEC 5-2

CBW instruction 3-108

CDQ instruction 3-228

CDQE instruction 3-108

CF (carry) flag, EFLAGS register 3-27, 3-85, 3-87, 3-89, 3-91, 3-109, 3-116, 3-233, 3-391, 3-396, 3-589, 4-278, 4-339, 4-351, 4-354, 4-375, 4-387

CLC instruction 3-109

CLD instruction 3-110

CLFLUSH instruction 3-111

CPUID flag 3-173

CLI instruction 3-113

CLTS instruction 3-115

CMC instruction 3-116

CMOVcc flag 3-173

CMOVcc instructions 3-117

CPUID flag 3-173

CMP instruction 3-121

CMPPD instruction 3-123

CMPPS instruction 3-130

CMPS instruction 3-136, 4-300

CMPSB instruction 3-136

CMPSD instruction 3-136, 3-140

CMPSQ instruction 3-136

CMPSS instruction 3-144

CMPSW instruction 3-136

CMPXCHG instruction 3-148, 3-468

CMPXCHG16B instruction 3-150

CPUID bit 3-171

CMPXCHG8B instruction 3-150

CPUID flag 3-173

COMISD instruction 3-153

COMISS instruction 3-155

Compatibility mode

introduction 2-6

see 64-bit mode

summary table notation 3-7

Compatibility, software 1-4

Condition code flags, EFLAGS register 3-117

## INDEX

- Condition code flags, x87 FPU status word  
  flags affected by instructions 3-13  
  setting 3-349, 3-351, 3-353
- Conditional jump 3-431
- Conforming code segment 3-445
- Constants (floating point), loading 3-303
- Control registers, moving values to and from 3-510
- Cosine, x87 FPU operation 3-279, 3-331
- CPL 3-113, 4-433
- CPUID instruction 3-157, 3-173  
  36-bit page size extension 3-173  
  AP-485 1-7  
  APIC on-chip 3-173  
  basic CPUID information 3-158  
  cache and TLB characteristics 3-158  
  CLFLUSH flag 3-173  
  CLFLUSH instruction cache line size 3-169  
  CMPXCHG16B flag 3-171  
  CMPXCHG8B flag 3-173  
  CPL qualified debug store 3-170  
  debug extensions, CR4.DE 3-173  
  debug store supported 3-174  
  deterministic cache parameters leaf 3-158, 3-160, 3-161, 3-162, 3-163, 3-164  
  extended function information 3-164  
  feature information 3-172  
  FPU on-chip 3-173  
  FSAVE flag 3-174  
  FXRSTOR flag 3-174  
  IA-32e mode available 3-165  
  input limits for EAX 3-166  
  L1 Context ID 3-171  
  local APIC physical ID 3-169  
  machine check architecture 3-173  
  machine check exception 3-173  
  memory type range registers 3-173  
  MONITOR feature information 3-178  
  MONITOR/MWAIT flag 3-170  
  MONITOR/MWAIT leaf 3-159, 3-160, 3-161, 3-162  
  MWAIT feature information 3-178  
  page attribute table 3-173  
  page size extension 3-173  
  performance monitoring features 3-179  
  physical address bits 3-166  
  physical address extension 3-173  
  power management 3-178, 3-179  
  processor brand index 3-169, 3-179  
  processor brand string 3-165, 3-179  
  processor serial number 3-158, 3-173  
  processor type field 3-169  
  RDMSR flag 3-173  
  returned in EBX 3-169  
  returned in ECX & EDX 3-169  
  self snoop 3-174  
  SpeedStep technology 3-170  
  SS2 extensions flag 3-174  
  SSE extensions flag 3-174  
  SSE3 extensions flag 3-170  
  SSSE3 extensions flag 3-170
- SYSENTER flag 3-173
- SYSEXIT flag 3-173
- thermal management 3-178, 3-179
- thermal monitor 3-170, 3-174
- time stamp counter 3-173
- using CPUID 3-157
- vendor ID string 3-166
- version information 3-158, 3-178
- virtual 8086 Mode flag 3-173
- virtual address bits 3-166
- WRMSR flag 3-173
- CQO instruction 3-228
- CRO control register 4-367
- CS register 3-96, 3-404, 3-423, 3-437, 3-506, 4-183
- CVTDQ2PD instruction 3-188
- CVTDQ2PS instruction 3-193
- CVTPD2DQ instruction 3-195
- CVTPD2PI instruction 3-197
- CVTPD2PS instruction 3-198
- CVTPI2PD instruction 3-200
- CVTPI2PS instruction 3-201
- CVTPS2DQ instruction 3-202, 3-203
- CVTPS2PD instruction 3-204
- CVTPS2PI instruction 3-206
- CVTSD2SI instruction 3-207
- CVTSD2SS instruction 3-209
- CVTSI2SD instruction 3-211
- CVTSI2SS instruction 3-213
- CVTSS2SD instruction 3-215
- CVTSS2SI instruction 3-216
- CVTPD2DQ instruction 3-218
- CVTPD2PI instruction 3-218, 3-220
- CVTPS2DQ instruction 3-221
- CVTPS2PI instruction 3-223
- CVTSD2SI instruction 3-224
- CVTSS2SI instruction 3-226
- CWD instruction 3-228
- CWDE instruction 3-108
- D**
- D (default operation size) flag, segment descriptor 4-187
- DAA instruction 3-229
- DAS instruction 3-231
- Debug registers, moving value to and from 3-513
- DEC instruction 3-233, 3-468
- Denormalized finite number 3-353
- Detecting and Enabling SMX  
  level 2 5-1
- DF (direction) flag, EFLAGS register 3-110, 3-137, 3-398, 3-470, 3-561, 4-19, 4-341, 4-376
- Displacement (operand addressing) 2-3
- DIV instruction 3-235
- Divide error exception (#DE) 3-235
- DIVPD instruction 3-238
- DIVPS instruction 3-240
- DIVSD instruction 3-242
- DIVSS instruction 3-243
- DS register 3-136, 3-451, 3-470, 3-561, 4-19
- E**
- EDI register 4-341, 4-376, 4-380

Effective address 3-455  
**EFLAGS** register  
 condition codes 3-119, 3-271, 3-276  
 flags affected by instructions 3-13  
 popping 4-191  
 popping on return from interrupt 3-423  
 pushing 4-272  
 pushing on interrupts 3-404  
 saving 4-329  
 status flags 3-121, 3-434, 4-346, 4-407  
**EIP** register 3-96, 3-404, 3-423, 3-437  
**EMMS** instruction 3-249  
**Encodings**  
 See machine instructions, opcodes  
**ENTER** instruction 3-251  
**GETSEC** 5-3, 5-9  
**ES** register 3-451, 4-19, 4-341, 4-380  
**ESI** register 3-136, 3-470, 3-561, 4-19, 4-376  
**ESP** register 3-96, 4-183  
**Exceptions**  
 BOUND range exceeded (#BR) 3-78, 4-424  
 notation 1-5  
 overflow exception (#OF) 3-404  
 returning from 3-423  
**GETSEC** 5-3, 5-5  
 Exponent, extracting from floating-point number 3-368  
 Extract exponent and significand, x87 FPU operation 3-368  
**F**  
**F2XM1** instruction 3-256, 3-368  
**FABS** instruction 3-258  
**FADD** instruction 3-260  
**FADDP** instruction 3-260  
 Far pointer, loading 3-451  
 Far return, RET instruction 4-302  
**FBLD** instruction 3-263  
**FBSTP** instruction 3-265  
**FCHS** instruction 3-267  
**FCLEX** instruction 3-269  
**FCMOVcc** instructions 3-271  
**FCOM** instruction 3-273  
**FCOMI** instruction 3-276  
**FCOMIP** instruction 3-276  
**FCOMP** instruction 3-273  
**FCOMPP** instruction 3-273  
**FCOS** instruction 3-279  
**FDECSTP** instruction 3-281  
**FDIV** instruction 3-282  
**FDIVP** instruction 3-282  
**FDIVR** instruction 3-285  
**FDIVRP** instruction 3-285  
 Feature information, processor 3-157  
**FFREE** instruction 3-288  
**FIADD** instruction 3-260  
**FICOM** instruction 3-289  
**FICOMP** instruction 3-289  
**FIDIV** instruction 3-282  
**FIDIVR** instruction 3-285  
**FILD** instruction 3-291  
**FIMUL** instruction 3-309

**FINCSTP** instruction 3-293  
**FINIT** instruction 3-294  
**FINIT/FNINIT** instructions 3-324  
**FIST** instruction 3-296  
**FISTP** instruction 3-296  
**FISTTP** instruction 3-299  
**FISUB** instruction 3-343  
**FISUBR** instruction 3-346  
**FLD** instruction 3-301  
**FLD1** instruction 3-303  
**FLDCW** instruction 3-305  
**FLDENV** instruction 3-307  
**FLDL2E** instruction 3-303  
**FLDL2T** instruction 3-303  
**FLDLG2** instruction 3-303  
**FLDLN2** instruction 3-303  
**FLDPI** instruction 3-303  
**FLDZ** instruction 3-303  
 Floating point instructions  
 machine encodings B-64  
 Floating-point exceptions  
 SSE and SSE2 SIMD 3-15  
 x87 FPU 3-15  
**Flushing**  
 caches 3-416, 4-547  
 TLB entry 3-418  
**FMUL** instruction 3-309  
**FMULP** instruction 3-309  
**FNCLEX** instruction 3-269  
**FNINIT** instruction 3-294  
**FNOP** instruction 3-312  
**FNSAVE** instruction 3-324  
**FNSTCW** instruction 3-337  
**FNSTENV** instruction 3-307, 3-339  
**FNSTSW** instruction 3-341  
**FPATAN** instruction 3-313  
**FPREM** instruction 3-315  
**FPREM1** instruction 3-317  
**FPTAN** instruction 3-319  
**FRNDINT** instruction 3-321  
**FRSTOR** instruction 3-322  
**FS** register 3-451  
**FSAVE** instruction 3-324  
**FSAVE/FNSAVE** instructions 3-322  
**FSCALE** instruction 3-327  
**FSIN** instruction 3-329  
**FSINCOS** instruction 3-331  
**FSQRT** instruction 3-333  
**FST** instruction 3-335  
**FSTCW** instruction 3-337  
**FSTENV** instruction 3-339  
**FSTP** instruction 3-335  
**FSTS** instruction 3-341  
**FSUB** instruction 3-343  
**FSUBP** instruction 3-343  
**FSUBR** instruction 3-346  
**FSUBRP** instruction 3-346  
**FTST** instruction 3-349  
**FUCOM** instruction 3-351

## INDEX

- FUCOMI instruction 3-276  
FUCOMIP instruction 3-276  
FUCOMP instruction 3-351  
FUCOMPP instruction 3-351  
FXAM instruction 3-353  
FXCH instruction 3-355  
FXRSTOR instruction 3-357  
    CPUID flag 3-174  
FXSAVE instruction 3-360, 4-542, 4-544, 4-567, 4-577, 4-581,  
4-585, 4-588, 4-591, 4-594, 4-597  
    CPUID flag 3-174  
FXTRACT instruction 3-327, 3-368  
FYL2X instruction 3-370  
FYL2XP1 instruction 3-372
- G**  
GDT (global descriptor table) 3-461, 3-464  
GDTR (global descriptor table register) 3-461, 4-349  
General-purpose instructions  
    64-bit encodings B-18  
    non-64-bit encodings B-7  
General-purpose registers  
    moving value to and from 3-506  
    popping all 4-187  
    pushing all 4-270  
GETSEC 5-1, 5-2, 5-5  
GS register 3-451
- H**  
HADDPD instruction 3-374, 3-375  
HADDPS instruction 3-377  
Hexadecimal numbers 1-5  
HLT instruction 3-380  
HSUBPD instruction 3-381  
HSUBPS instruction 3-384
- I**  
IA-32e mode  
    CPUID flag 3-165  
    introduction 2-6, 2-12  
    see 64-bit mode  
    see compatibility mode  
IDIV instruction 3-387  
IDT (interrupt descriptor table) 3-404, 3-461  
IDTR (interrupt descriptor table register) 3-461, 4-363  
IF (interrupt enable) flag, EFLAGS register 3-113, 4-377  
Immediate operands 2-3  
IMUL instruction 3-390  
IN instruction 3-394  
INC instruction 3-396, 3-468  
Index (operand addressing) 2-3  
Initialization x87 FPU 3-294  
initiating logical processor 5-3, 5-4, 5-5, 5-9, 5-20  
INS instruction 3-398, 4-300  
INSB instruction 3-398  
INSD instruction 3-398  
instruction encodings B-60, B-66, B-73  
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  
    operands 1-4  
    prefixes 2-1  
    r/m field 2-3  
    reg(opcode 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 3-398  
INT 3 instruction 3-404  
Integer, storing, x87 FPU data type 3-296  
Intel 64 architecture  
    definition of 1-3  
    instruction format 2-1  
    relation to IA-32 1-3  
Intel developer link 1-7  
Intel NetBurst microarchitecture 1-2  
Intel software network link 1-7  
Intel VTune Performance Analyzer  
    related information 1-7  
Intel Xeon processor 1-1  
Intel® Trusted Execution Technology 5-3  
Inter-privilege level  
    call, CALL instruction 3-95  
    return, RET instruction 4-302  
Interrupts  
    interrupt vector 4 3-404  
    returning from 3-423  
    software 3-404  
INTn instruction 3-404  
INTO instruction 3-404  
Intrinsics  
    compiler functional equivalents C-1  
    composite C-14  
    description of 3-11  
    list of C-1  
    simple C-2  
INVD instruction 3-416  
INVLPG instruction 3-418  
IOPL (I/O privilege level) field, EFLAGS register 3-113, 4-272,  
4-377  
IRET instruction 3-423  
IRETD instruction 3-423
- J**  
Jcc instructions 3-431  
JMP instruction 3-436  
Jump operation 3-436
- L**  
L1 Context ID 3-171  
LAHF instruction 3-444  
LAR instruction 3-445  
Last branch  
    interrupt & exception recording  
    description of 4-314

LDDQU instruction 3-448  
 LDMXCSR instruction 3-450, 4-285, 4-549  
 LDS instruction 3-451  
 LDT (local descriptor table) 3-464  
 LDTR (local descriptor table register) 3-464, 4-365  
 LEA instruction 3-455  
 LEAVE instruction 3-458  
 LES instruction 3-451  
 LFENCE instruction 3-460  
 LFS instruction 3-451  
 LGDT instruction 3-461  
 LGS instruction 3-451  
 LIDT instruction 3-461  
 LLDT instruction 3-464  
 LMSW instruction 3-466  
 Load effective address operation 3-455  
 LOCK prefix 3-25, 3-28, 3-51, 3-87, 3-89, 3-91, 3-148, 3-233, 3-396, 3-468, 4-6, 4-9, 4-11, 4-339, 4-387, 4-558, 4-563, 4-571  
 Locking operation 3-468  
 LODS instruction 3-470, 4-300  
 LODSB instruction 3-470  
 LODSD instruction 3-470  
 LODSQ instruction 3-470  
 LODSW instruction 3-470  
 Log (base 2), x87 FPU operation 3-372  
 Log epsilon, x87 FPU operation 3-370  
 LOOP instructions 3-473  
 LOOPcc instructions 3-473  
 LSL instruction 3-475  
 LSS instruction 3-451  
 LTR instruction 3-478  
 LZCNT - Count the Number of Leading Zero Bits 3-480

**M**

Machine check architecture  
 CPUID flag 3-173  
 description 3-173

Machine instructions  
 64-bit mode B-1  
 condition test (tttn) field B-5  
 direction bit (d) field B-6  
 floating-point instruction encodings B-64  
 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-2, B-3  
 REX prefixes B-2  
 segment register (sreg) field B-4  
 sign-extend (s) bit B-4  
 SIMD 64-bit encodings B-37  
 special 64-bit encodings B-64  
 special fields B-2  
 special-purpose register (eee) field B-5  
 SSE encodings B-42-B-48  
 SSE2 encodings B-48-B-58

SSE3 encodings B-59-B-60  
 SSSE3 encodings B-60-B-63  
 VMX encodings B-117, B-118  
 See also: opcodes

Machine status word, CR0 register 3-466, 4-367  
 MASKMOVQDQU instruction 3-513  
 MASKMOVQ instruction 4-507  
 MAXPD instruction 3-486  
 MAXPS instruction 3-488  
 MAXSD instruction 3-490  
 MAXSS instruction 3-492  
 measured environment 5-1  
 Measured Launched Environment 5-1, 5-24  
 MFENCE instruction 3-494  
 MINPD instruction 3-495  
 MINPS instruction 3-497  
 MINSD instruction 3-499  
 MINSS instruction 3-501  
 MLE 5-1  
 MMX instructions  
 CPUID flag for technology 3-174  
 encodings B-38

Mod field, instruction format 2-3  
 Model & family information 3-178  
 ModR/M byte 2-3  
 16-bit addressing forms 2-4  
 32-bit addressing forms of 2-5  
 description of 2-3

MONITOR instruction 3-503  
 CPUID flag 3-170  
 feature data 3-178

MOV instruction 3-505  
 MOV instruction (control registers) 3-510  
 MOV instruction (debug registers) 3-513, 3-519  
 MOVAPD instruction 3-515  
 MOVAPS instruction 3-517  
 MOVD instruction 3-519  
 MOVDUP instruction 3-524  
 MOVDQ2Q instruction 3-530  
 MOVDQA instruction 3-526  
 MOVDQU instruction 3-528  
 MOVHLPS instruction 3-531  
 MOVHPD instruction 3-533  
 MOVHPS instruction 3-535  
 MOVLHP instruction 3-537  
 MOVLHPS instruction 3-537  
 MOVLPD instruction 3-539  
 MOVLPS instruction 3-541  
 MOVMSKPD instruction 3-543  
 MOVMSKPS instruction 3-545  
 MOVNTDQ instruction 3-560  
 MOVNTI instruction 3-560  
 MOVNTPD instruction 3-553  
 MOVNTPS instruction 3-555  
 MOVNTQ instruction 3-557  
 MOVQ instruction 3-519, 3-558  
 MOVQ2DQ instruction 3-560  
 MOVS instruction 3-561, 4-300  
 MOVSb instruction 3-561

## INDEX

- MOVSD instruction 3-561, 3-565
- MOVSHDUP instruction 3-567
- MOVSLDUP instruction 3-569
- MOVSQ instruction 3-561
- MOVSS instruction 3-571
- MOVSW instruction 3-561
- MOVSX instruction 3-573
- MOVSD instruction 3-573
- MOVUPD instruction 3-575
- MOVUPS instruction 3-577
- MOVZX instruction 3-579
- MSRs (model specific registers)
  - reading 4-287
- MUL instruction 3-20, 3-589
- MULPD instruction 3-591
- MULPS instruction 3-593
- MULSD instruction 3-595
- MULSS instruction 3-596
- Multi-byte no operation 4-6, 4-8, B-12
- MULX - Unsigned Multiply Without Affecting Flags 3-597
- MVMM 5-1, 5-4, 5-5, 5-36
- MWAIT instruction 3-599
  - CPUID flag 3-170
  - feature data 3-178
- N**
  - NaN. testing for 3-349
  - Near
    - return, RET instruction 4-302
  - NEG instruction 3-468, 4-6
  - NetBurst microarchitecture (see Intel NetBurst microarchitecture)
  - No operation 4-6, 4-8, B-12
  - Nomenclature, used in instruction reference pages 3-1
  - NOP instruction 4-8
  - NOT instruction 3-468, 4-9
- Notation
  - bit and byte order 1-3
  - exceptions 1-5
  - hexadecimal and binary numbers 1-5
  - instruction operands 1-4
  - reserved bits 1-4
  - segmented addressing 1-5
- Notational conventions 1-3
- NT (nested task) flag, EFLAGS register 3-423
- O**
  - OF (carry) flag, EFLAGS register 3-391
  - OF (overflow) flag, EFLAGS register 3-27, 3-404, 3-589, 4-339, 4-351, 4-354, 4-387
  - Opcode format 2-2
  - Opcodes
    - addressing method codes for A-1
    - extensions A-18
    - extensions tables A-19
    - group numbers A-18
    - integers
      - one-byte opcodes A-8
      - two-byte opcodes A-8
    - key to abbreviations A-1
    - look-up examples A-3, A-18, A-21
    - ModR/M byte A-18
- one-byte opcodes A-3, A-8
- 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-8
- VMX instructions B-117, B-118
- x87 ESC instruction opcodes A-21
- Operands 1-4
- OR instruction 3-468, 4-11
- ORPD instruction 4-13
- ORPS instruction 4-15
- OUT instruction 4-17
- OUTS instruction 4-19, 4-300
- OUTSB instruction 4-19
- OUTSD instruction 4-19
- OUTSW instruction 4-19
- Overflow exception (#OF) 3-404
- P**
  - P6 family processors
    - description of 1-1
    - machine encodings B-41
  - PABSB instruction 4-23
  - PABSD instruction 4-23
  - PABSW instruction 4-23
  - PACKSSDW instruction 4-27
  - PACKSSWB instruction 4-27
  - PACKUSWB instruction 4-35
  - PADDB instruction 4-38
  - PADDD instruction 4-38
  - PADDQ instruction 4-42
  - PADDSB instruction 4-44
  - PADDSW instruction 4-44
  - PADDUSB instruction 4-47
  - PADDUSW instruction 4-47
  - PADDW instruction 4-38
  - PALIGNR instruction 4-50
  - PAND instruction 4-53
  - PANDN instruction 4-55
  - GETSEC 5-4
  - PAUSE instruction 4-57
  - PAVGB instruction 4-58
  - PAVGW instruction 4-58
  - PCE flag, CR4 register 4-290
  - PCMPEQB instruction 4-71
  - PCMPEQD instruction 4-71
  - PCMPEQW instruction 4-71
  - PCMPGTB instruction 4-81
  - PCMPGTD instruction 4-81
  - PCMPGTW instruction 4-81
  - PDEP - Parallel Bits Deposit 4-91
  - PE (protection enable) flag, CR0 register 3-466
  - Pending break enable 3-174
  - Pentium 4 processor 1-1
  - Pentium II processor 1-2
  - Pentium III processor 1-2
  - Pentium Pro processor 1-2
  - Pentium processor 1-1
  - Pentium processor family processors

- machine encodings B-37
- Performance-monitoring counters
  - CPUID inquiry for 3-179
- PEXT - Parallel Bits Extract 4-93
- PEXTRW instruction 4-98
- PHADD instruction 4-101
- PHADDSW instruction 4-105
- PHADDW instruction 4-101
- PHSUBD instruction 4-109
- PHSUBSW instruction 4-112
- PHSUBW instruction 4-109
- Pi 3-303
- PINSRW instruction 4-116, 4-213
- PMADDUDSW instruction 4-118
- PMADDUDSW instruction 4-118
- PMADDWD instruction 4-120
- PMAXSW instruction 4-128
- PMAXUB instruction 4-131
- PMINSW instruction 4-143
- PMINUB instruction 4-146
- PMOVMSKB instruction 4-149
- PMULHRSW instruction 4-165
- PMULHUW instruction 4-168
- PMULHW instruction 4-172
- PMULLW instruction 4-177
- PMULUDQ instruction 4-180
- POP instruction 4-182
- POPA instruction 4-187
- POPAD instruction 4-187
- POPF instruction 4-191
- POPDF instruction 4-191
- POPFQ instruction 4-191
- POR instruction 4-194
- PREFETCHh instruction 4-196
- 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, 3-468
  - Operand-size override prefix 2-2
  - REP or REPE/REPZ 2-1
  - REP/REPE/REPZ/REPNE/REPNZ 4-299
  - REPNE/REPNZ 2-1
  - REX prefix encodings B-2
  - Segment override prefixes 2-1
- PSADBW instruction 4-200
- PSHUFB instruction 4-203
- PSHUFD instruction 4-206
- PSHUFHW instruction 4-208
- PSHUFLW instruction 4-210
- PSHUFW instruction 4-212
- PSIGNB instruction 4-213
- PSIGND instruction 4-213
- PSIGNW instruction 4-213
- PSLLD instruction 4-219
- PSLLDQ instruction 4-217
- PSLLQ instruction 4-219
- PSLLW instruction 4-219
- PSRAD instruction 4-225
- PSRAW instruction 4-225
- PSRLD instruction 4-232
- PSRLDQ instruction 4-230
- PSRLQ instruction 4-232
- PSRLW instruction 4-232
- PSUBB instruction 4-238
- PSUBD instruction 4-238
- PSUBQ instruction 4-243
- PSUBSB instruction 4-245
- PSUBSW instruction 4-245
- PSUBUSB instruction 4-248
- PSUBUSW instruction 4-248
- PSUBW instruction 4-238
- PUNPCKHBW instruction 4-253
- PUNPCKHDQ instruction 4-253
- PUNPCKHQDQ instruction 4-253
- PUNPCKHWD instruction 4-253
- PUNPCKLBW instruction 4-260
- PUNPCKLDO instruction 4-260
- PUNPCKLQDQ instruction 4-260
- PUNPCKLWD instruction 4-260
- PUSH instruction 4-267
- PUSHA instruction 4-270
- PUSHAD instruction 4-270
- PUSHF instruction 4-272
- PUSHFD instruction 4-272
- PXOR instruction 4-274
- R**
  - R/m field, instruction format 2-3
  - RC (rounding control) field, x87 FPU control word 3-296, 3-303, 3-335
  - RCL instruction 4-276
  - RCPPS instruction 4-281
  - RCPSS instruction 4-283
  - RCR instruction 4-276
  - RDMSR instruction 4-287, 4-290, 4-295
    - CPUID flag 3-173
  - RDPMC instruction 4-289
  - RDTSC instruction 4-293, 4-295, 4-297
  - Reg/opcode field, instruction format 2-3
  - Related literature 1-6
  - Remainder, x87 FPU operation 3-317
  - REP/REPE/REPZ/REPNE/REPNZ prefixes 3-137, 3-399, 4-20, 4-299
  - Reserved
    - use of reserved bits 1-4
  - Responding logical processor 5-4
  - responding logical processor 5-4, 5-5
  - RET instruction 4-302
  - REX prefixes
    - addressing modes 2-8
    - and INC/DEC 2-7
    - encodings 2-7, B-2
    - field names 2-8
    - ModR/M byte 2-7
    - overview 2-7
    - REXB 2-7

## INDEX

- REX.R 2-7
- REX.W 2-7
- special encodings 2-10
- RIP-relative addressing 2-11
- ROL instruction 4-276
- ROR instruction 4-276
- RORX - Rotate Right Logical Without Affecting Flags 4-312
- Rounding
  - modes, floating-point operations 4-314
- Rounding control (RC) field
  - MXCSR register 4-314
  - x87 FPU control word 4-314
- Rounding, round to integer, x87 FPU operation 3-321
- RPL field 3-62
- RSM instruction 4-323
- RSQRTPS instruction 4-325
- RSQRTSS instruction 4-327
- S**
  - Safer Mode Extensions 5-1
  - SAHF instruction 4-329
  - SAL instruction 4-331
  - SAR instruction 4-331
  - SBB instruction 3-468, 4-338
  - Scale (operand addressing) 2-3
  - Scale, x87 FPU operation 3-327
  - Scan string instructions 4-341
  - SCAS instruction 4-300, 4-341
  - SCASB instruction 4-341
  - SCASD instruction 4-341
  - SCASW instruction 4-341
  - Segment
    - descriptor, segment limit 3-475
    - limit 3-475
    - registers, moving values to and from 3-506
    - selector, RPL field 3-62
  - Segmented addressing 1-5
  - Self Snoop 3-174
  - GETSEC 5-2, 5-3, 5-5
  - SENTER sleep state 5-9
  - SETcc instructions 4-345
  - GETSEC 5-4
  - SF (sign) flag, EFLAGS register 3-27
  - SFENCE instruction 4-348
  - SGDT instruction 4-349
  - SHAF instruction 4-329
  - Shift instructions 4-331
  - SHL instruction 4-331
  - SHLD instruction 4-351
  - SHR instruction 4-331
  - SHRD instruction 4-354
  - SHUFPD instruction 4-357
  - SHUFPS instruction 4-360
  - SIB byte 2-3
    - 32-bit addressing forms of 2-6, 2-19
    - description of 2-3
  - SIDT instruction 4-349, 4-363
  - Significand, extracting from floating-point number 3-368
  - SIMD floating-point exceptions, unmasking, effects of 3-450, 4-285, 4-549
  - Sine, x87 FPU operation 3-329, 3-331
  - SINIT 5-4
  - SLDT instruction 4-365
  - GETSEC 5-4
  - SMSW instruction 4-367
  - SpeedStep technology 3-170
  - SQRTPD instruction 4-369
  - SQRTPS instruction 4-371
  - SQRTSD instruction 4-373
  - SQRTSS instruction 4-374
  - Square root, Fx87 PU operation 3-333
  - SS register 3-451, 3-506, 4-183
  - SSE extensions
    - cacheability instruction encodings B-48
    - CPUID flag 3-174
    - floating-point encodings B-42
    - instruction encodings B-42
    - integer instruction encodings B-46
    - memory ordering encodings B-48
  - SSE2 extensions
    - cacheability instruction encodings B-58
    - CPUID flag 3-174
    - floating-point encodings B-49
    - integer instruction encodings B-54
  - SSE3
    - CPUID flag 3-170
  - SSE3 extensions
    - CPUID flag 3-170
    - event mgmt instruction encodings B-59
    - floating-point instruction encodings B-59
    - integer instruction encodings B-60
  - SSSE3 extensions B-60, B-66, B-73
    - CPUID flag 3-170
  - Stack, pushing values on 4-267
  - Status flags, EFLAGS register 3-119, 3-121, 3-271, 3-276, 3-434, 4-346, 4-407
  - STC instruction 4-375
  - STD instruction 4-376
  - Stepping information 3-178
  - STI instruction 4-377
  - STMCSR instruction 4-379
  - STOS instruction 4-300, 4-380
  - STOSB instruction 4-380
  - STOSD instruction 4-380
  - STOSQ instruction 4-380
  - STOSW instruction 4-380
  - STR instruction 4-384
  - String instructions 3-136, 3-398, 3-470, 3-561, 4-19, 4-341, 4-380
  - SUB instruction 3-22, 3-231, 3-468, 4-386
  - SUBPD instruction 4-388
  - SUBSS instruction 4-393
  - SWAPGS instruction 4-394
  - SYSCALL instruction 4-396
  - SYSENTER instruction 4-398
    - CPUID flag 3-173
  - SYSEXIT instruction 4-401
    - CPUID flag 3-173
  - SYSRET instruction 4-404

- T**
- Tangent, x87 FPU operation 3-319
  - Task register
    - loading 3-478
    - storing 4-384
  - Task switch
    - CALL instruction 3-95
    - return from nested task, IRET instruction 3-423
  - TEST instruction 4-407, 4-539
  - Thermal Monitor
    - CPUID flag 3-174
  - Thermal Monitor 2 3-170
    - CPUID flag 3-170
  - Time Stamp Counter 3-173
  - Time-stamp counter, reading 4-295, 4-297
  - TLB entry, invalidating (flushing) 3-418
  - Trusted Platform Module 5-4, 5-5
  - TS (task switched) flag, CR0 register 3-115
  - TSD flag, CR4 register 4-295, 4-297
  - TSS, relationship to task register 4-384
  - TZCNT - Count the Number of Trailing Zero Bits 4-409
- U**
- UCOMISD instruction 4-409
  - UCOMISS instruction 4-413
  - UD2 instruction 4-415
  - Undefined, format opcodes 3-349
  - Unordered values 3-273, 3-349, 3-351
  - UNPCKHPD instruction 4-416
  - UNPCKHPS instruction 4-418
  - UNPCKLPD instruction 4-420
  - UNPCKLPS instruction 4-422
- V**
- VERR instruction 4-433
  - Version information, processor 3-157
  - VERW instruction 4-433
  - VEX 3-3
    - VEX.B 3-3
    - VEX.L 3-3
    - VEX.mmmmm 3-3
    - VEX.pp 3-4
    - VEX.R 3-4
    - VEX.vvvv 3-3
    - VEX.W 3-3
    - VEX.X 3-3
  - VEXTRACTI128 - Extract packed Integer Values 4-522
  - VFMADD132PD/VFMADD213PD/VFMADD231PD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values 4-432
  - VFMADD132SD/VFMADD213SD/VFMADD231SD - Fused Multiply-Add of Scalar Double-Precision Floating-Point Values 4-443
  - VFMADD132SS/VFMADD213SS/VFMADD231SS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values 4-445
  - VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD - Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values 4-447
  - VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS - Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values 4-450
  - VFMSUB132PD/VFMSUB213PD/VFMSUB231PD - Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values 4-459
  - VFMSUB132PS/VFMSUB213PS/VFMSUB231PS - Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values 4-462
  - VFMSUB132SD/VFMSUB213SD/VFMSUB231SD - Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values 4-465
  - VFMSUB132SS/VFMSUB213SS/VFMSUB231SS - Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values 4-467
  - VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values 4-453
  - VFNMADD132PD/VFNMADD213PD/VFNMADD231PD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values 4-469
  - VFNMADD132PS/VFNMADD213PS/VFNMADD231PS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values 4-472
  - VFNMADD132SD/VFNMADD213SD/VFNMADD231SD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values 4-475
  - VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD - Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values 4-479
  - VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values 4-485
  - VGATHERDPS/VGATHERQPS - Gather Packed SP FP values Using Signed Dword/Qword Indices 4-493
  - Virtual Machine Monitor 5-1
  - VM (virtual 8086 mode) flag, EFLAGS register 3-423
  - VMM 5-1
  - VPBROADCAST - Broadcast Integer Data 4-512
  - VPERM2I128 - Permute Integer Values 4-520
  - VPERMD - Full Doublewords Element Permutation 4-516
  - VPERMPD - Permute Double-Precision Floating-Point Elements 4-517
  - VPERMPS - Permute Single-Precision Floating-Point Elements 4-518
  - VPERMQ - Qwords Element Permutation 4-519
  - VPGATHERDQ/VPGATHERQQ - Gather Packed Qword values Using Signed Dword/Qword Indices 4-501
  - VPSRLVD/VPSRLVQ - Variable Bit Shift Right Logical 4-537
- W**
- WAIT/FWAIT instructions 4-546
  - GETSEC 5-4
  - WBINVD instruction 4-547
  - WBINVD/INV bit 3-159
  - Write-back and invalidate caches 4-547
  - WRMSR instruction 4-551
    - CPUID flag 3-173
- X**
- x87 FPU
    - checking for pending x87 FPU exceptions 4-546
    - constants 3-303
    - initialization 3-294
    - instruction opcodes A-21
  - x87 FPU control word

## INDEX

loading 3-305, 3-307  
RC field 3-296, 3-303, 3-335  
restoring 3-322  
saving 3-324, 3-339  
storing 3-337  
x87 FPU data pointer 3-307, 3-322, 3-324, 3-339  
x87 FPU instruction pointer 3-307, 3-322, 3-324, 3-339  
x87 FPU last opcode 3-307, 3-322, 3-324, 3-339  
x87 FPU status word  
    condition code flags 3-273, 3-289, 3-349, 3-351, 3-353  
    loading 3-307  
    restoring 3-322  
    saving 3-324, 3-339, 3-341  
    TOP field 3-293  
        x87 FPU flags affected by instructions 3-13  
x87 FPU tag word 3-307, 3-322, 3-324, 3-339  
XABORT - Transaction Abort 4-556  
XADD instruction 3-468, 4-558  
XCHG instruction 3-468, 4-563  
XEND - Transaction End 4-565  
XFEATURE\_ENABLED\_MASK 4-567, 4-597, 4-598  
XGETBV 4-567, 4-577, 4-581, B-41  
XLAB instruction 4-569  
XLAT instruction 4-569  
XOR instruction 3-468, 4-571  
XORPD instruction 4-573  
XORPS instruction 4-575  
XRSTOR B-41  
XSAVE 4-567, 4-579, 4-580, 4-582, 4-583, 4-585, 4-586, 4-587,  
    4-588, 4-589, 4-590, 4-591, 4-592, 4-593, 4-594, 4-595, 4-596,  
    4-597, 4-598, B-41  
XSETBV 4-591, 4-597, B-41  
XTEST - Test If In Transactional Execution 4-599

## Z

ZF (zero) flag, EFLAGS register 3-148, 3-445, 3-473, 3-475,  
4-300, 4-433