

# IA-32 Intel Architecture Software Developer's Manual

## Volume 2: Instruction Set Reference

**NOTE:** The *IA-32 Intel Architecture Software Developer's Manual* consists of three volumes: *Basic Architecture*, Order Number 245470; *Instruction Set Reference*, Order Number 245471; and the *System Programming Guide*, Order Number 245472. Please refer to all three volumes when evaluating your design needs.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

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

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.

Intel's IA-32 Intel® Architecture processors (e.g., Pentium® 4 and Pentium® III processors) may contain design defects or errors known as errata. Current characterized errata are available on request.

Intel®, Intel386™, Intel486™, Pentium®, Intel® NetBurst™, MMX™, and Itanium™ are trademarks owned by Intel Corporation.

\*Third-party brands and names are the property of their respective owners.

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

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

Intel Corporation  
P.O. Box 7641  
Mt. Prospect IL 60056-7641

or call 1-800-879-4683  
or visit Intel's website at <http://www.intel.com>



# TABLE OF CONTENTS

	PAGE
<b>CHAPTER 1</b>	
<b>ABOUT THIS MANUAL</b>	
1.1.	IA-32 PROCESSORS COVERED IN THIS MANUAL . . . . . 1-1
1.2.	OVERVIEW OF THE IA-32 INTEL® ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 2: INSTRUCTION SET REFERENCE . . . . . 1-1
1.3.	OVERVIEW OF THE IA-32 INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 1: BASIC ARCHITECTURE . . . . . 1-2
1.4.	OVERVIEW OF THE IA-32 INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 3: SYSTEM PROGRAMMING GUIDE . . . . . 1-4
1.5.	NOTATIONAL CONVENTIONS . . . . . 1-6
1.5.1.	Bit and Byte Order . . . . . 1-6
1.5.2.	Reserved Bits and Software Compatibility . . . . . 1-6
1.5.3.	Instruction Operands . . . . . 1-7
1.5.4.	Hexadecimal and Binary Numbers . . . . . 1-7
1.5.5.	Segmented Addressing . . . . . 1-8
1.5.6.	Exceptions . . . . . 1-8
1.6.	RELATED LITERATURE . . . . . 1-9
<b>CHAPTER 2</b>	
<b>INSTRUCTION FORMAT</b>	
2.1.	GENERAL INSTRUCTION FORMAT . . . . . 2-1
2.2.	INSTRUCTION PREFIXES . . . . . 2-1
2.3.	OPCODE . . . . . 2-3
2.4.	MODR/M AND SIB BYTES . . . . . 2-3
2.5.	DISPLACEMENT AND IMMEDIATE BYTES . . . . . 2-3
2.6.	ADDRESSING-MODE ENCODING OF MODR/M AND SIB BYTES . . . . . 2-4
<b>CHAPTER 3</b>	
<b>INSTRUCTION SET REFERENCE</b>	
3.1.	INTERPRETING THE INSTRUCTION REFERENCE PAGES . . . . . 3-1
3.1.1.	Instruction Format . . . . . 3-1
3.1.1.1.	Opcode Column . . . . . 3-1
3.1.1.2.	Instruction Column . . . . . 3-2
3.1.1.3.	Description Column . . . . . 3-5
3.1.1.4.	Description . . . . . 3-5
3.1.2.	Operation . . . . . 3-5
3.1.3.	Intel C/C++ Compiler Intrinsic Equivalents . . . . . 3-8
3.1.3.1.	The Intrinsic API . . . . . 3-8
3.1.3.2.	MMX™ Technology Intrinsic . . . . . 3-8
3.1.3.3.	SSE and SSE2 Intrinsic . . . . . 3-9
3.1.4.	Flags Affected . . . . . 3-11
3.1.5.	FPU Flags Affected . . . . . 3-11
3.1.6.	Protected Mode Exceptions . . . . . 3-11
3.1.7.	Real-Address Mode Exceptions . . . . . 3-12
3.1.8.	Virtual-8086 Mode Exceptions . . . . . 3-13
3.1.9.	Floating-Point Exceptions . . . . . 3-13
3.1.10.	SIMD Floating-Point Exceptions . . . . . 3-13
3.2.	INSTRUCTION REFERENCE . . . . . 3-14

	PAGE
AAA—ASCII Adjust After Addition . . . . .	3-15
AAD—ASCII Adjust AX Before Division . . . . .	3-16
AAM—ASCII Adjust AX After Multiply . . . . .	3-17
AAS—ASCII Adjust AL After Subtraction . . . . .	3-18
ADC—Add with Carry . . . . .	3-19
ADD—Add . . . . .	3-21
ADDPD—Add Packed Double-Precision Floating-Point Values . . . . .	3-23
ADDPS—Add Packed Single-Precision Floating-Point Values . . . . .	3-25
ADDSD—Add Scalar Double-Precision Floating-Point Values . . . . .	3-27
ADDSS—Add Scalar Single-Precision Floating-Point Values . . . . .	3-29
AND—Logical AND . . . . .	3-31
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values . . . . .	3-33
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values . . . . .	3-35
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values . . . . .	3-37
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values . . . . .	3-39
ARPL—Adjust RPL Field of Segment Selector . . . . .	3-41
BOUND—Check Array Index Against Bounds . . . . .	3-43
BSF—Bit Scan Forward . . . . .	3-45
BSR—Bit Scan Reverse . . . . .	3-47
BSWAP—Byte Swap . . . . .	3-49
BT—Bit Test . . . . .	3-50
BTC—Bit Test and Complement . . . . .	3-52
BTR—Bit Test and Reset . . . . .	3-54
BTS—Bit Test and Set . . . . .	3-56
CALL—Call Procedure . . . . .	3-58
CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword . . . . .	3-69
CDQ—Convert Double to Quad . . . . .	3-70
CLC—Clear Carry Flag . . . . .	3-71
CLD—Clear Direction Flag . . . . .	3-72
CLFLUSH—Flush Cache Line . . . . .	3-73
CLI—Clear Interrupt Flag . . . . .	3-75
CLTS—Clear Task-Switched Flag in CR0 . . . . .	3-77
CMC—Complement Carry Flag . . . . .	3-78
CMOVcc—Conditional Move . . . . .	3-79
CMP—Compare Two Operands . . . . .	3-83
CMPPD—Compare Packed Double-Precision Floating-Point Values . . . . .	3-85
CMPPS—Compare Packed Single-Precision Floating-Point Values . . . . .	3-89
CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands . . . . .	3-93
CMPSD—Compare Scalar Double-Precision Floating-Point Values . . . . .	3-96
CMPSS—Compare Scalar Single-Precision Floating-Point Values . . . . .	3-100
CMPXCHG—Compare and Exchange . . . . .	3-104
CMPXCHG8B—Compare and Exchange 8 Bytes . . . . .	3-106
COMISD—Compare Scalar Ordered Double-Precision Floating-Point	

	<b>PAGE</b>
Values and Set EFLAGS . . . . .	3-108
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS . . . . .	3-111
CPUID—CPU Identification . . . . .	3-114
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values. . . . .	3-128
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values . . . . .	3-130
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers . . . . .	3-132
CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers . . . . .	3-134
CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values . . . . .	3-136
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values. . . . .	3-138
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values . . . . .	3-140
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers . . . . .	3-142
CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values . . . . .	3-144
CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers . . . . .	3-146
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer . . . . .	3-148
CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value. . . . .	3-150
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value. . . . .	3-152
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value. . . . .	3-154
CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value . . . . .	3-156
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer . . . . .	3-158
CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers . . . . .	3-160
CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers . . . . .	3-162
CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers . . . . .	3-164
CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers . . . . .	3-166
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer . . . . .	3-168
CVTTSS2SI—Convert with Truncation Scalar Single-Precision	

	PAGE
Floating-Point Value to Doubleword Integer . . . . .	3-170
CWD/CDQ—Convert Word to Doubleword/Convert Doubleword to Quadword	3-172
CWDE—Convert Word to Doubleword . . . . .	3-173
DAA—Decimal Adjust AL after Addition . . . . .	3-174
DAS—Decimal Adjust AL after Subtraction . . . . .	3-176
DEC—Decrement by 1 . . . . .	3-177
DIV—Unsigned Divide . . . . .	3-179
DIVPD—Divide Packed Double-Precision Floating-Point Values . . . . .	3-182
DIVPS—Divide Packed Single-Precision Floating-Point Values . . . . .	3-184
DIVSD—Divide Scalar Double-Precision Floating-Point Values . . . . .	3-186
DIVSS—Divide Scalar Single-Precision Floating-Point Values . . . . .	3-188
EMMS—Empty MMX State . . . . .	3-190
ENTER—Make Stack Frame for Procedure Parameters . . . . .	3-191
F2XM1—Compute $2x^{-1}$ . . . . .	3-194
FABS—Absolute Value . . . . .	3-196
FADD/FADDP/FIADD—Add . . . . .	3-198
FBLD—Load Binary Coded Decimal . . . . .	3-201
FBSTP—Store BCD Integer and Pop . . . . .	3-203
FCFS—Change Sign . . . . .	3-206
FCLEX/FNCLEX—Clear Exceptions . . . . .	3-208
FCMOVcc—Floating-Point Conditional Move . . . . .	3-210
FCOM/FCOMP/FCOMPP—Compare Floating Point Values . . . . .	3-212
FCOMI/FCOMIP/ FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS . . . . .	3-215
FCOS—Cosine . . . . .	3-218
FDECSTP—Decrement Stack-Top Pointer . . . . .	3-220
FDIV/FDIVP/FIDIV—Divide . . . . .	3-221
FDIVR/FDIVRP/FIDIVR—Reverse Divide . . . . .	3-225
FFREE—Free Floating-Point Register . . . . .	3-229
FICOM/FICOMP—Compare Integer . . . . .	3-230
FILD—Load Integer . . . . .	3-232
FINCSTP—Increment Stack-Top Pointer . . . . .	3-234
FINIT/FNINIT—Initialize Floating-Point Unit . . . . .	3-235
FIST/FISTP—Store Integer . . . . .	3-237
FLD—Load Floating Point Value . . . . .	3-240
FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant . . . . .	3-242
FLDCW—Load x87 FPU Control Word . . . . .	3-244
FLDENV—Load x87 FPU Environment . . . . .	3-246
FMUL/FMULP/FIMUL—Multiply . . . . .	3-248
FNOP—No Operation . . . . .	3-251
FPATAN—Partial Arctangent . . . . .	3-252
FPREM—Partial Remainder . . . . .	3-254
FPREM1—Partial Remainder . . . . .	3-257
FPTAN—Partial Tangent . . . . .	3-260
FRNDINT—Round to Integer . . . . .	3-262
FRSTOR—Restore x87 FPU State . . . . .	3-263
FSAVE/FNSAVE—Store x87 FPU State . . . . .	3-265

	<b>PAGE</b>
FSCALE—Scale . . . . .	3-268
FSIN—Sine . . . . .	3-270
FSINCOS—Sine and Cosine . . . . .	3-272
FSQRT—Square Root . . . . .	3-274
FST/FSTP—Store Floating Point Value . . . . .	3-276
FSTCW/FNSTCW—Store x87 FPU Control Word . . . . .	3-279
FSTENV/FNSTENV—Store x87 FPU Environment . . . . .	3-281
FSTSW/FNSTSW—Store x87 FPU Status Word . . . . .	3-284
FSUB/FSUBP/FISUB—Subtract . . . . .	3-287
FSUBR/FSUBRP/FISUBR—Reverse Subtract . . . . .	3-290
FTST—TEST . . . . .	3-293
FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values . . . . .	3-295
FWAIT—Wait . . . . .	3-298
FXAM—Examine . . . . .	3-299
FXCH—Exchange Register Contents . . . . .	3-301
FXRSTOR—Restore x87 FPU, MMX, SSE, and SSE2 State . . . . .	3-303
FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State . . . . .	3-305
FXTRACT—Extract Exponent and Significand . . . . .	3-311
FYL2X—Compute $y * \log_2 x$ . . . . .	3-313
FYL2XP1—Compute $y * \log_2(x + 1)$ . . . . .	3-315
HLT—Halt . . . . .	3-317
IDIV—Signed Divide . . . . .	3-318
IMUL—Signed Multiply . . . . .	3-321
IN—Input from Port . . . . .	3-324
INC—Increment by 1 . . . . .	3-326
INS/INSB/INSW/INSD—Input from Port to String . . . . .	3-328
INT n/INTO/INT 3—Call to Interrupt Procedure . . . . .	3-331
INVD—Invalidate Internal Caches . . . . .	3-343
INVLPG—Invalidate TLB Entry . . . . .	3-345
IRET/IRETD—Interrupt Return . . . . .	3-346
Jcc—Jump if Condition Is Met . . . . .	3-354
JMP—Jump . . . . .	3-358
LAHF—Load Status Flags into AH Register . . . . .	3-365
LAR—Load Access Rights Byte . . . . .	3-366
LDMXCSR—Load MXCSR Register . . . . .	3-369
LDS/LES/LFS/LGS/LSS—Load Far Pointer . . . . .	3-371
LEA—Load Effective Address . . . . .	3-374
LEAVE—High Level Procedure Exit . . . . .	3-376
LES—Load Full Pointer . . . . .	3-378
LFENCE—Load Fence . . . . .	3-379
LFS—Load Full Pointer . . . . .	3-380
LGDT/LIDT—Load Global/Interrupt Descriptor Table Register . . . . .	3-381
LGS—Load Full Pointer . . . . .	3-383
LLDT—Load Local Descriptor Table Register . . . . .	3-384
LIDT—Load Interrupt Descriptor Table Register . . . . .	3-386
LMSW—Load Machine Status Word . . . . .	3-387
LOCK—Assert LOCK# Signal Prefix . . . . .	3-389



	PAGE
LODS/LODSB/LODSW/LODSD—Load String . . . . .	3-391
LOOP/LOOPcc—Loop According to ECX Counter . . . . .	3-394
LSL—Load Segment Limit . . . . .	3-396
LSS—Load Full Pointer . . . . .	3-399
LTR—Load Task Register . . . . .	3-400
MASKMOVDQU—Store Selected Bytes of Double Quadword . . . . .	3-402
MASKMOVQ—Store Selected Bytes of Quadword . . . . .	3-404
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values . . . . .	3-407
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values . . . . .	3-410
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value . . . . .	3-413
MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value . . . . .	3-416
MFENCE—Memory Fence . . . . .	3-419
MINPD—Return Minimum Packed Double-Precision Floating-Point Values . . . . .	3-420
MINPS—Return Minimum Packed Single-Precision Floating-Point Values . . . . .	3-423
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value . . . . .	3-426
MINSS—Return Minimum Scalar Single-Precision Floating-Point Value . . . . .	3-429
MOV—Move . . . . .	3-432
MOV—Move to/from Control Registers . . . . .	3-437
MOV—Move to/from Debug Registers . . . . .	3-439
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values . . . . .	3-441
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values . . . . .	3-443
MOVD—Move Doubleword . . . . .	3-445
MOVDQA—Move Aligned Double Quadword . . . . .	3-447
MOVDQU—Move Unaligned Double Quadword . . . . .	3-449
MOVDQ2Q—Move Quadword from XMM to MMX Register . . . . .	3-451
MOVHLPs— Move Packed Single-Precision Floating-Point Values High to Low . . . . .	3-452
MOVHPD—Move High Packed Double-Precision Floating-Point Value . . . . .	3-453
MOVHPS—Move High Packed Single-Precision Floating-Point Values . . . . .	3-455
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High . . . . .	3-457
MOVLPD—Move Low Packed Double-Precision Floating-Point Value . . . . .	3-458
MOVLPS—Move Low Packed Single-Precision Floating-Point Values . . . . .	3-460
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask . . . . .	3-462
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask . . . . .	3-464
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint . . . . .	3-466
MOVNTI—Store Doubleword Using Non-Temporal Hint . . . . .	3-468
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint . . . . .	3-470
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint . . . . .	3-472
MOVNTQ—Store of Quadword Using Non-Temporal Hint . . . . .	3-474
MOVQ—Move Quadword . . . . .	3-476
MOVQ2DQ—Move Quadword from MMX to XMM Register . . . . .	3-478
MOVS/MOVSb/MOVSsw/MOVSd—Move Data from String to String . . . . .	3-479
MOVSD—Move Scalar Double-Precision Floating-Point Value . . . . .	3-482
MOVSS—Move Scalar Single--Precision Floating-Point Values . . . . .	3-485



	<b>PAGE</b>
MOVSX—Move with Sign-Extension . . . . .	3-488
MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values . . . . .	3-490
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values . . . . .	3-492
MOVZX—Move with Zero-Extend . . . . .	3-494
MUL—Unsigned Multiply . . . . .	3-496
MULPD—Multiply Packed Double-Precision Floating-Point Values . . . . .	3-498
MULPS—Multiply Packed Single-Precision Floating-Point Values . . . . .	3-500
MULSD—Multiply Scalar Double-Precision Floating-Point Values . . . . .	3-502
MULSS—Multiply Scalar Single-Precision Floating-Point Values . . . . .	3-504
NEG—Two's Complement Negation . . . . .	3-506
NOP—No Operation . . . . .	3-508
NOT—One's Complement Negation . . . . .	3-509
OR—Logical Inclusive OR . . . . .	3-511
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values . . . . .	3-513
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values . . . . .	3-515
OUT—Output to Port . . . . .	3-517
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port . . . . .	3-519
PACKSSWB/PACKSSDW—Pack with Signed Saturation . . . . .	3-522
PACKUSWB—Pack with Unsigned Saturation . . . . .	3-526
PADDB/PADDW/PADDD—Add Packed Integers . . . . .	3-529
PADDQ—Add Packed Quadword Integers . . . . .	3-532
PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation . . . . .	3-534
PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation . . . . .	3-537
PAND—Logical AND . . . . .	3-540
PANDN—Logical AND NOT . . . . .	3-542
PAUSE—Spin Loop Hint . . . . .	3-544
PAVGB/PAVGW—Average Packed Integers . . . . .	3-545
PCMPEQB/PCMPEQW/PCMPEQD—Compare Packed Data for Equal . . . . .	3-548
PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than . . . . .	3-552
PEXTRW—Extract Word . . . . .	3-556
PINSRW—Insert Word . . . . .	3-558
PMADDWD—Multiply and Add Packed Integers . . . . .	3-561
PMAXSW—Maximum of Packed Signed Word Integers . . . . .	3-564
PMAXUB—Maximum of Packed Unsigned Byte Integers . . . . .	3-567
PMINSW—Minimum of Packed Signed Word Integers . . . . .	3-570
PMINUB—Minimum of Packed Unsigned Byte Integers . . . . .	3-573
PMOVBMSKB—Move Byte Mask . . . . .	3-576
PMULHUW—Multiply Packed Unsigned Integers and Store High Result . . . . .	3-578
PMULHW—Multiply Packed Signed Integers and Store High Result . . . . .	3-581
PMULLW—Multiply Packed Signed Integers and Store Low Result . . . . .	3-584
PMULUDQ—Multiply Packed Unsigned Doubleword Integers . . . . .	3-587
POP—Pop a Value from the Stack . . . . .	3-589
POPA/POPAD—Pop All General-Purpose Registers . . . . .	3-593
POPF/POPFD—Pop Stack into EFLAGS Register . . . . .	3-595
POR—Bitwise Logical OR . . . . .	3-598

	PAGE
PREFETCHh—Prefetch Data Into Caches . . . . .	3-600
PSADBW—Compute Sum of Absolute Differences . . . . .	3-602
PSHUFD—Shuffle Packed Doublewords . . . . .	3-605
PSHUFHW—Shuffle Packed High Words . . . . .	3-608
PSHUFLW—Shuffle Packed Low Words . . . . .	3-610
PSHUFW—Shuffle Packed Words . . . . .	3-612
PSLLDQ—Shift Double Quadword Left Logical . . . . .	3-614
PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical . . . . .	3-615
PSRAW/PSRAD—Shift Packed Data Right Arithmetic . . . . .	3-620
PSRLDQ—Shift Double Quadword Right Logical . . . . .	3-624
PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical . . . . .	3-625
PSUBB/PSUBW/PSUBD—Subtract Packed Integers . . . . .	3-630
PSUBQ—Subtract Packed Quadword Integers . . . . .	3-634
PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation. . . . .	3-636
PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation . . . . .	3-639
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ—Unpack High Data. . . . .	3-642
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data . . . . .	3-646
PUSH—Push Word or Doubleword Onto the Stack . . . . .	3-650
PUSHA/PUSHAD—Push All General-Purpose Registers. . . . .	3-653
PUSHF/PUSHFD—Push EFLAGS Register onto the Stack . . . . .	3-655
PXOR—Logical Exclusive OR. . . . .	3-657
RCL/RCR/ROL/ROR—Rotate . . . . .	3-660
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values . . . . .	3-665
RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values . . . . .	3-667
RDMSR—Read from Model Specific Register . . . . .	3-669
RDPMC—Read Performance-Monitoring Counters . . . . .	3-671
RDTSC—Read Time-Stamp Counter . . . . .	3-673
REP/REPE/REPZ/REPNE /REPZ—Repeat String Operation Prefix . . . . .	3-674
RET—Return from Procedure . . . . .	3-677
ROL/ROR—Rotate . . . . .	3-683
RSM—Resume from System Management Mode . . . . .	3-684
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values . . . . .	3-685
RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value . . . . .	3-687
SAHF—Store AH into Flags . . . . .	3-689
SAL/SAR/SHL/SHR—Shift . . . . .	3-690
SBB—Integer Subtraction with Borrow . . . . .	3-694
SCAS/SCASB/SCASW/SCASD—Scan String . . . . .	3-696
SETcc—Set Byte on Condition . . . . .	3-699
SFENCE—Store Fence. . . . .	3-701

	<b>PAGE</b>
SGDT/SIDT—Store Global/Interrupt Descriptor Table Register . . . . .	3-702
SHL/SHR—Shift Instructions . . . . .	3-704
SHLD—Double Precision Shift Left . . . . .	3-705
SHRD—Double Precision Shift Right . . . . .	3-707
SHUFPS—Shuffle Packed Double-Precision Floating-Point Values . . . . .	3-709
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values . . . . .	3-712
SIDT—Store Interrupt Descriptor Table Register . . . . .	3-715
SLDT—Store Local Descriptor Table Register . . . . .	3-716
SMSW—Store Machine Status Word . . . . .	3-718
SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values . . . . .	3-720
SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values . . . . .	3-722
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value . . . . .	3-724
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value . . . . .	3-726
STC—Set Carry Flag . . . . .	3-728
STD—Set Direction Flag . . . . .	3-729
STI—Set Interrupt Flag . . . . .	3-730
STMXCSR—Store MXCSR Register State . . . . .	3-732
STOS/STOSB/STOSW/STOSD—Store String . . . . .	3-734
STR—Store Task Register . . . . .	3-737
SUB—Subtract . . . . .	3-739
SUBPD—Subtract Packed Double-Precision Floating-Point Values . . . . .	3-741
SUBPS—Subtract Packed Single-Precision Floating-Point Values . . . . .	3-743
SUBSD—Subtract Scalar Double-Precision Floating-Point Values . . . . .	3-745
SUBSS—Subtract Scalar Single-Precision Floating-Point Values . . . . .	3-747
SYSENTER—Fast System Call . . . . .	3-749
SYSEXIT—Fast Return from Fast System Call . . . . .	3-753
TEST—Logical Compare . . . . .	3-756
UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS . . . . .	3-758
UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS . . . . .	3-761
UD2—Undefined Instruction . . . . .	3-764
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values . . . . .	3-765
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values . . . . .	3-768
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values . . . . .	3-771
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values . . . . .	3-774
VERR, VERW—Verify a Segment for Reading or Writing . . . . .	3-777
WAIT/FWAIT—Wait . . . . .	3-779
WBINVD—Write Back and Invalidate Cache . . . . .	3-780

	PAGE
WRMSR—Write to Model Specific Register . . . . .	3-782
XADD—Exchange and Add. . . . .	3-784
XCHG—Exchange Register/Memory with Register . . . . .	3-786
XLAT/XLATB—Table Look-up Translation . . . . .	3-788
XOR—Logical Exclusive OR . . . . .	3-790
XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values . . . . .	3-792
XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values . . . . .	3-794

**APPENDIX A  
OPCODE MAP**

A.1. KEY TO ABBREVIATIONS . . . . .	A-1
A.1.1. Codes for Addressing Method . . . . .	A-1
A.1.2. Codes for Operand Type . . . . .	A-3
A.1.3. Register Codes . . . . .	A-3
A.2. OPCODE LOOK-UP EXAMPLES . . . . .	A-3
A.2.1. One-Byte Opcode Instructions . . . . .	A-4
A.2.2. Two-Byte Opcode Instructions . . . . .	A-4
A.2.3. Opcode Map Notes . . . . .	A-5
A.2.4. Opcode Extensions For One- And Two-byte Opcodes . . . . .	A-10
A.2.5. Escape Opcode Instructions . . . . .	A-12
A.2.5.1. Opcodes with ModR/M Bytes in the 00H through BFH Range . . . . .	A-12
A.2.5.2. Opcodes with ModR/M Bytes outside the 00H through BFH Range . . . . .	A-12
A.2.5.3. Escape Opcodes with D8 as First Byte . . . . .	A-12
A.2.5.4. Escape Opcodes with D9 as First Byte . . . . .	A-14
A.2.5.5. Escape Opcodes with DA as First Byte . . . . .	A-15
A.2.5.6. Escape Opcodes with DB as First Byte . . . . .	A-16
A.2.5.7. Escape Opcodes with DC as First Byte . . . . .	A-18
A.2.5.8. Escape Opcodes with DD as First Byte . . . . .	A-19
A.2.5.9. Escape Opcodes with DE as First Byte . . . . .	A-21
A.2.5.10. Escape Opcodes with DF As First Byte . . . . .	A-22

**APPENDIX B  
INSTRUCTION FORMATS AND ENCODINGS**

B.1. MACHINE INSTRUCTION FORMAT . . . . .	B-1
B.1.1. Reg Field (reg). . . . .	B-2
B.1.2. Encoding of Operand Size Bit (w) . . . . .	B-3
B.1.3. Sign Extend (s) Bit. . . . .	B-3
B.1.4. Segment Register Field (sreg). . . . .	B-4
B.1.5. Special-Purpose Register (eee) Field . . . . .	B-4
B.1.6. Condition Test Field (ttn) . . . . .	B-5
B.1.7. Direction (d) Bit . . . . .	B-5
B.2. GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS . . . . .	B-6
B.3. MMX INSTRUCTION FORMATS AND ENCODINGS . . . . .	B-19
B.3.1. Granularity Field (gg). . . . .	B-19
B.3.2. MMX and General-Purpose Register Fields (mmxreg and reg)B-19	
B.3.3. MMX Instruction Formats and Encodings Table . . . . .	B-19
B.4. P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS . . . . .	B-22
B.5. SSE INSTRUCTION FORMATS AND ENCODINGS . . . . .	B-23
B.6. SSE2 INSTRUCTION FORMATS AND ENCODINGS . . . . .	B-31
B.6.1. Granularity Field (gg). . . . .	B-31

	PAGE
B.7. FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS .....	B-45
<b>APPENDIX C</b>	
<b>INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS</b>	
C.1. SIMPLE INTRINSICS .....	C-3
C.2. COMPOSITE INTRINSICS .....	C-28

# TABLE OF FIGURES

	PAGE
Figure 1-1. Bit and Byte Order . . . . .	1-6
Figure 2-1. IA-32 Instruction Format . . . . .	2-1
Figure 3-1. Bit Offset for BIT[EAX,21]. . . . .	3-8
Figure 3-2. Memory Bit Indexing . . . . .	3-11
Figure 3-3. Version Information in the EAX Register . . . . .	3-116
Figure 3-4. Feature Information in the EDX Register . . . . .	3-118
Figure 3-5. Operation of the PACKSSDW Instruction Using 64-bit Operands. . . . .	3-522
Figure 3-6. PMADDWD Execution Model Using 64-bit Operands . . . . .	3-561
Figure 3-7. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands . . . . .	3-578
Figure 3-8. PMULLU Instruction Operation Using 64-bit Operands . . . . .	3-584
Figure 3-9. PSADBW Instruction Operation Using 64-bit Operands. . . . .	3-602
Figure 3-10. PSHUFD Instruction Operation. . . . .	3-605
Figure 3-11. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand . . . . .	3-615
Figure 3-12. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand . . . . .	3-620
Figure 3-13. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand . . . . .	3-625
Figure 3-14. PUNPCKHBW Instruction Operation Using 64-bit Operands. . . . .	3-642
Figure 3-15. PUNPCKLBW Instruction Operation Using 64-bit Operands . . . . .	3-646
Figure 3-16. SHUFPS Shuffle Operation . . . . .	3-709
Figure 3-17. SHUFPS Shuffle Operation . . . . .	3-712
Figure 3-18. UNPCKHPD Instruction High Unpack and Interleave Operation . . . . .	3-765
Figure 3-19. UNPCKHPS Instruction High Unpack and Interleave Operation . . . . .	3-768
Figure 3-20. UNPCKLPD Instruction Low Unpack and Interleave Operation . . . . .	3-771
Figure 3-21. UNPCKLPS Instruction Low Unpack and Interleave Operation . . . . .	3-774
Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3). . . . .	A-10
Figure B-1. General Machine Instruction Format . . . . .	B-1

# TABLE OF TABLES

	PAGE
Table 2-1.	16-Bit Addressing Forms with the ModR/M Byte . . . . . 2-5
Table 2-2.	32-Bit Addressing Forms with the ModR/M Byte . . . . . 2-6
Table 2-3.	32-Bit Addressing Forms with the SIB Byte . . . . . 2-7
Table 3-1.	Register Encodings Associated with the +rb, +rw, and +rd Nomenclature. . . . . 3-2
Table 3-2.	IA-32 General Exceptions. . . . . 3-12
Table 3-3.	x87 FPU Floating-Point Exceptions . . . . . 3-13
Table 3-4.	SIMD Floating-Point Exceptions. . . . . 3-14
Table 3-5.	Comparison Predicate for CMPDD and CMPPS Instructions. . . . . 3-85
Table 3-6.	Information Returned by CPUID Instruction . . . . . 3-115
Table 3-7.	Highest CPUID Source Operand for IA-32 Processors . . . . . 3-116
Table 3-8.	Processor Type Field . . . . . 3-117
Table 3-9.	CPUID Feature Flags Returned in EDX Register . . . . . 3-119
Table 3-10.	Encoding of Cache and TLB Descriptors . . . . . 3-121
Table 3-11.	Mapping of Brand Indices and IA-32 Processor Brand Strings . . . . . 3-124
Table 3-12.	Processor Brand String Returned with First Pentium 4 Processor. . . . . 3-125
Table 3-13.	Layout of FXSAVE and FXRSTOR Memory Region . . . . . 3-305
Table 3-14.	MSRs Used By the SYSENTER and SYSEXIT Instructions . . . . . 3-749
Table A-1.	Notes on Instruction Set Encoding Tables . . . . . A-5
Table A-2.	One-byte Opcode Map (Left) . . . . . A-6
Table A-3.	One-byte Opcode Map (Right) . . . . . A-7
Table A-4.	Two-byte Opcode Map (Left) (First Byte is OFH) . . . . . A-8
Table A-5.	Two-byte Opcode Map (Right) (First Byte is OFH). . . . . A-9
Table A-6.	Opcode Extensions for One- and Two-byte Opcodes by Group Number. . . . . A-11
Table A-7.	D8 Opcode Map When ModR/M Byte is Within 00H to BFH1 . . . . . A-12
Table A-8.	D8 Opcode Map When ModR/M Byte is Outside 00H to BFH1 . . . . . A-13
Table A-9.	D9 Opcode Map When ModR/M Byte is Within 00H to BFH1. . . . . A-14
Table A-10.	D9 Opcode Map When ModR/M Byte is Outside 00H to BFH1 . . . . . A-15
Table A-11.	DA Opcode Map When ModR/M Byte is Within 00H to BFH1 . . . . . A-15
Table A-12.	DA Opcode Map When ModR/M Byte is Outside 00H to BFH1 . . . . . A-16
Table A-13.	DB Opcode Map When ModR/M Byte is Within 00H to BFH1 . . . . . A-17
Table A-14.	DB Opcode Map When ModR/M Byte is Outside 00H to BFH1 . . . . . A-17
Table A-15.	DC Opcode Map When ModR/M Byte is Within 00H to BFH1 . . . . . A-18
Table A-16.	DC Opcode Map When ModR/M Byte is Outside 00H to BFH4 . . . . . A-19
Table A-17.	DD Opcode Map When ModR/M Byte is Within 00H to BFH1 . . . . . A-20
Table A-18.	DD Opcode Map When ModR/M Byte is Outside 00H to BFH1 . . . . . A-20
Table A-19.	DE Opcode Map When ModR/M Byte is Within 00H to BFH1 . . . . . A-21
Table A-20.	DE Opcode Map When ModR/M Byte is Outside 00H to BFH1 . . . . . A-22
Table A-21.	DF Opcode Map When ModR/M Byte is Within 00H to BFH1 . . . . . A-23
Table A-22.	DF Opcode Map When ModR/M Byte is Outside 00H to BFH1 . . . . . A-23
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-2
Table B-3.	Encoding of reg Field When w Field is Present in Instruction. . . . . B-3
Table B-4.	Encoding of Operand Size (w) Bit. . . . . B-3
Table B-5.	Encoding of Sign-Extend (s) Bit . . . . . B-3
Table B-6.	Encoding of the Segment Register (sreg) Field . . . . . B-4
Table B-7.	Encoding of Special-Purpose Register (eee) Field. . . . . B-4
Table B-8.	Encoding of Conditional Test (ttn) Field. . . . . B-5
Table B-9.	Encoding of Operation Direction (d) Bit . . . . . B-6
Table B-10.	General Purpose Instruction Formats and Encodings . . . . . B-6



	PAGE
Table B-11. Encoding of Granularity of Data Field (gg) . . . . .	B-19
Table B-12. MMX Instruction Formats and Encodings. . . . .	B-19
Table B-13. Formats and Encodings of P6 Family Instructions . . . . .	B-22
Table B-14. Formats and Encodings of SSE SIMD Floating-Point Instructions. . . . .	B-23
Table B-15. Formats and Encodings of SSE SIMD Integer Instructions . . . . .	B-29
Table B-16. Format and Encoding of the SSE Cacheability and Memory Ordering Instructions . . . . .	B-30
Table B-17. Encoding of Granularity of Data Field (gg) . . . . .	B-31
Table B-18. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions . . . . .	B-31
Table B-19. Formats and Encodings of the SSE2 SIMD Integer Instructions . . . . .	B-38
Table B-20. Format and Encoding of the SSE2 Cacheability Instructions. . . . .	B-44
Table B-21. General Floating-Point Instruction Formats . . . . .	B-45
Table B-22. Floating-Point Instruction Formats and Encodings. . . . .	B-46
Table C-1. Simple Intrinsics . . . . .	C-3
Table C-2. Composite Intrinsics . . . . .	C-28





intel®

1

# About This Manual





# CHAPTER 1

## ABOUT THIS MANUAL

The *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference* (Order Number 245471) is part of a three-volume set that describes the architecture and programming environment of all IA-32 Intel® Architecture processors. The other two volumes in this set are:

- The *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture* (Order Number 245470).
- The *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide* (Order Number 245472).

The *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of an IA-32 processor; the *IA-32 Intel Architecture Software Developer's Manual, Volume 2*, describes the instructions set of the processor and the opcode structure. These two volumes are aimed at application programmers who are writing programs to run under existing operating systems or executives. The *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, describes the operating-system support environment of an IA-32 processor, including memory management, protection, task management, interrupt and exception handling, and system management mode. It also provides IA-32 processor compatibility information. This volume is aimed at operating-system and BIOS designers and programmers.

### 1.1. IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual includes information pertaining primarily to the most recent IA-32 processors, which include the Pentium® processor, the P6 family processors, and the Pentium® 4 processors. The P6 family processors are those IA-32 processors based on the P6 family micro-architecture. This family includes the Pentium® Pro, Pentium® II, and Pentium® III processors. The Pentium 4 processor is the first of a family of IA-32 processors based on the new Intel® NetBurst™ micro-architecture.

### 1.2. OVERVIEW OF THE IA-32 INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 2: INSTRUCTION SET REFERENCE

The contents of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2* are as follows:

**Chapter 1 — About This Manual.** Gives an overview of all three volumes of the *IA-32 Intel Architecture 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.** Describes each of the 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. The general-purpose, x87 FPU, Intel MMX™ technology, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), and system instructions are included in this chapter.

**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 Intrinsic and Functional Equivalents.** Lists the Intel C/C++ compiler intrinsics and their assembly code equivalents for each of the IA-32 MMX, SSE, and SSE2 instructions.

## 1.3. OVERVIEW OF THE IA-32 INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 1: BASIC ARCHITECTURE

The contents of this manual are as follows:

**Chapter 1 — About This Manual.** Gives an overview of all three volumes of the *IA-32 Intel Architecture 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 — Introduction to the IA-32 Architecture.** Introduces the IA-32 architecture and the families of Intel processors that are based on this architecture. It also gives an overview of the common features found in these processors and brief history of the IA-32 architecture.

**Chapter 3 — Basic Execution Environment.** Introduces the models of memory organization and describes the register set used by applications.

**Chapter 4 — Data Types.** Describes the data types and addressing modes recognized by the processor; provides an overview of real numbers and floating-point formats and of floating-point exceptions.

**Chapter 5 — Instruction Set Summary.** Lists all the IA-32 architecture instructions, divided into technology groups (general-purpose, x87 FPU, MMX technology, SSE, SSE2, and system instructions). Within these groups, the instructions are presented in functionally related groups.

**Chapter 6 — Procedure Calls, Interrupts, and Exceptions.** Describes the procedure stack and the mechanisms provided for making procedure calls and for servicing interrupts and exceptions.

**Chapter 7 — Programming With the General-Purpose and System Instructions.** Describes the basic load and store, program control, arithmetic, and string instructions that operate on basic data types and on the general-purpose and segment registers; describes the system instructions that are executed in protected mode.

**Chapter 8 — Programming With the x87 Floating Point Unit.** Describes the x87 floating-point unit (FPU), including the floating-point registers and data types; gives an overview of the floating-point instruction set; and describes the processor's floating-point exception conditions.

**Chapter 9 — Programming with Intel MMX Technology.** Describes the Intel MMX technology, including MMX registers and data types, and gives an overview of the MMX instruction set.

**Chapter 10 — Programming with Streaming SIMD Extensions (SSE).** Describes the SSE extensions, including the XMM registers, the MXCSR register, and the packed single-precision floating-point data types; gives an overview of the SSE instruction set; and gives guidelines for writing code that accesses the SSE extensions.

**Chapter 11 — Programming with Streaming SIMD Extensions 2 (SSE2).** Describes the SSE2 extensions, including XMM registers and the packed double-precision floating-point data types; gives an overview of the SSE2 instruction set; and gives guidelines for writing code that accesses the SSE2 extensions. This chapter also describes the SIMD floating-point exceptions that can be generated with SSE and SSE2 instructions, and it gives general guidelines for incorporating support for the SSE and SSE2 extensions into operating system and applications code.

**Chapter 12 — Input/Output.** Describes the processor's I/O mechanism, including I/O port addressing, the I/O instructions, and the I/O protection mechanism.

**Chapter 13 — Processor Identification and Feature Determination.** Describes how to determine the CPU type and the features that are available in the processor.

**Appendix A — EFLAGS Cross-Reference.** Summarizes how the IA-32 instructions affect the flags in the EFLAGS register.

**Appendix B — EFLAGS Condition Codes.** Summarizes how the conditional jump, move, and byte set on condition code instructions use the condition code flags (OF, CF, ZF, SF, and PF) in the EFLAGS register.

**Appendix C — Floating-Point Exceptions Summary.** Summarizes the exceptions that can be raised by the x87 FPU floating-point and the SSE and SSE2 SIMD floating-point instructions.

**Appendix D — Guidelines for Writing x87 FPU Exception Handlers.** Describes how to design and write MS-DOS\* compatible exception handling facilities for FPU exceptions, including both software and hardware requirements and assembly-language code examples. This appendix also describes general techniques for writing robust FPU exception handlers.

**Appendix E — Guidelines for Writing SIMD Floating-Point Exception Handlers.** Gives guidelines for writing exception handlers to handle exceptions generated by the SSE and SSE2 SIMD floating-point instructions.

## 1.4. OVERVIEW OF THE IA-32 INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 3: SYSTEM PROGRAMMING GUIDE

The contents of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3* are as follows:

**Chapter 1 — About This Manual.** Gives an overview of all three volumes of the *IA-32 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 — System Architecture Overview.** Describes the modes of operation of an IA-32 processor and the mechanisms provided in the IA-32 architecture to support operating systems and executives, including the system-oriented registers and data structures and the system-oriented instructions. The steps necessary for switching between real-address and protected modes are also identified.

**Chapter 3 — Protected-Mode Memory Management.** Describes the data structures, registers, and instructions that support segmentation and paging and explains how they can be used to implement a “flat” (unsegmented) memory model or a segmented memory model.

**Chapter 4 — Protection.** Describes the support for page and segment protection provided in the IA-32 architecture. This chapter also explains the implementation of privilege rules, stack switching, pointer validation, user and supervisor modes.

**Chapter 5 — Interrupt and Exception Handling.** Describes the basic interrupt mechanisms defined in the IA-32 architecture, shows how interrupts and exceptions relate to protection, and describes how the architecture handles each exception type. Reference information for each IA-32 exception is given at the end of this chapter.

**Chapter 6 — Task Management.** Describes the mechanisms that the IA-32 architecture provides to support multitasking and inter-task protection.

**Chapter 7 — Multiple Processor Management.** Describes the instructions and flags that support multiple processors with shared memory, memory ordering, and the advanced programmable interrupt controller (APIC).

**Chapter 8 — Processor Management and Initialization.** Defines the state of an IA-32 processor after reset initialization. This chapter also explains how to set up an IA-32 processor for real-address mode operation and protected mode operation, and how to switch between modes.

**Chapter 9 — Memory Cache Control.** Describes the general concept of caching, the caching mechanisms supported by the IA-32 architecture, and the cache control instructions. This chapter also describes the memory type range registers (MTRRs) and how they can be used to map memory types of physical memory.

**Chapter 10 — Intel MMX Technology System Programming.** Describes those aspects of the Intel MMX technology that must be handled and considered at the system programming level, including task switching, exception handling, and compatibility with existing system environments.

**Chapter 11 — SSE and SSE2 System Programming.** Describes those aspects of SSE and SSE2 extensions that must be handled and considered at the system programming level, including task switching, exception handling, and compatibility with existing system environments.

**Chapter 12 — System Management.** Describes the IA-32 architecture's system management mode (SMM) and the thermal monitoring facilities.

**Chapter 13 — Machine-Check Architecture.** Describes the machine-check architecture.

**Chapter 14 — Debugging and Performance Monitoring.** Describes the debugging registers and other debug mechanism provided in the IA-32 architecture. This chapter also describes the time-stamp counter and the performance-monitoring counters.

**Chapter 15 — 8086 Emulation.** Describes the real-address and virtual-8086 modes of the IA-32 architecture.

**Chapter 16 — Mixing 16-Bit and 32-Bit Code.** Describes how to mix 16-bit and 32-bit code modules within the same program or task.

**Chapter 17 — IA-32 Architecture Compatibility.** Describes the programming among the IA-32 processors, which include the Intel 286, Intel386™, Intel486™, Pentium, P6 family, and Pentium 4 processors. The P6 family includes the Pentium Pro, Pentium II, and Pentium III processors. The Pentium 4 processor is the first of a family of IA-32 processors based on the new Intel NetBurst micro-architecture. The differences among the 32-bit IA-32 processors are also described throughout the three volumes of the *IA-32 Software Developer's Manual*, as relevant to particular features of the architecture. This chapter provides a collection of all the relevant compatibility information for all IA-32 processors and also describes the basic differences with respect to the 16-bit IA-32 processors (the Intel 8086 and Intel 286 processors).

**Appendix A — Performance-Monitoring Events.** Lists the events that can be counted with the performance-monitoring counters and the codes used to select these events.

**Appendix B — Model Specific Registers (MSRs).** Lists the MSRs available in the Pentium, P6 family, and Pentium 4 processors and their functions.

**Appendix C — Dual-Processor (DP) Bootup Sequence Example (Specific to Pentium Processors).** Gives an example of how to use the DP protocol to boot two Pentium processors (a primary processor and a secondary processor) in a DP system and initialize their APICs.

**Appendix D — Multiple-Processor (MP) Bootup Sequence Example (Specific to P6 Family Processors).** Gives an example of how to use of the MP protocol to boot two P6 family processors in a multiple-processor (MP) system and initialize their APICs.

**Appendix E — Programming the LINT0 and LINT1 Inputs.** Gives an example of how to program the LINT0 and LINT1 pins for specific interrupt vectors.

**Appendix E — Interpreting Machine-Check Error Codes.** Gives an example of how to interpret the error codes for a machine-check error that occurred on a P6 family processor.

**Appendix F — APIC Bus Message Formats.** Describes the message formats for messages transmitted on the APIC bus for P6 family and Pentium processors.

## 1.5. NOTATIONAL CONVENTIONS

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

### 1.5.1. 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.

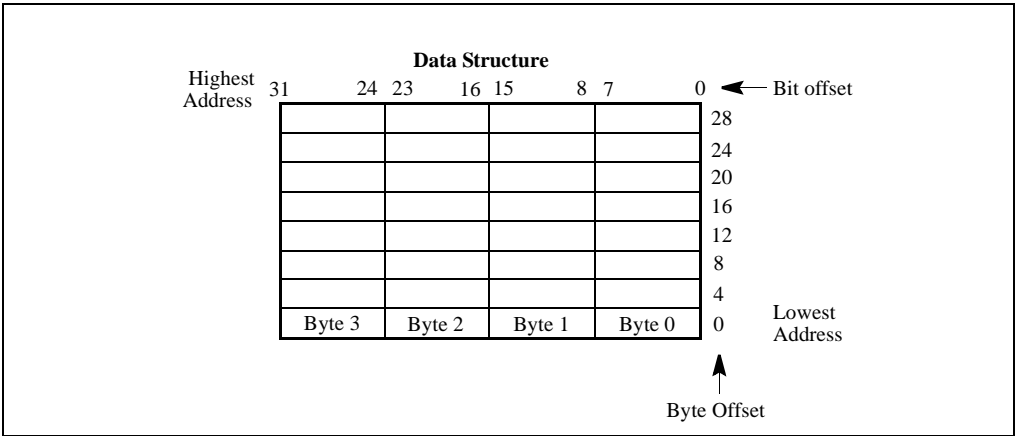


Figure 1-1. Bit and Byte Order

### 1.5.2. 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.

### 1.5.3. 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.

### 1.5.4. 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.

### 1.5.5. 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
```

### 1.5.6. 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)
```

See Chapter 5, *Interrupt and Exception Handling*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for a list of exception mnemonics and their descriptions.

## 1.6. RELATED LITERATURE

Literature related to IA-32 processors is listed on-line at the following Intel web site:

<http://developer.intel.com/design/processor/>

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

- Data Sheet for a particular Intel IA-32 processor.
- Specification Update for a particular Intel IA-32 processor.
- AP-485, *Intel Processor Identification and the CPUID Instruction*, Order Number 241618.
- *Intel Pentium 4 Optimization Reference Manual*, Order Number 248966.



intel®

2

# Instruction Format



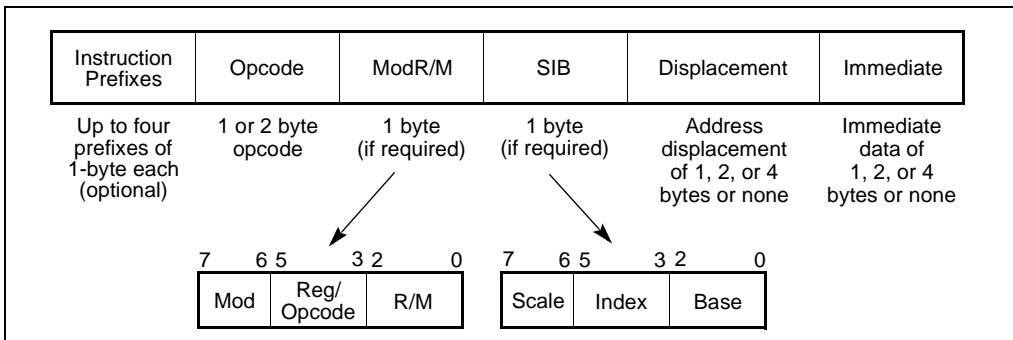


# CHAPTER 2 INSTRUCTION FORMAT

This chapter describes the instruction format for all IA-32 processors.

## 2.1. GENERAL INSTRUCTION FORMAT

All IA-32 instruction encodings are subsets of the general instruction format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), one or two primary opcode 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. IA-32 Instruction Format**

## 2.2. INSTRUCTION PREFIXES

The instruction prefixes are divided into four groups, each with a set of allowable prefix codes:

- Group 1
  - Lock and repeat prefixes:
    - F0H—LOCK.
    - F2H—REPNE/REP NZ (used only with string instructions).
    - F3H—REP (use only with string instructions).
    - F3H—REPE/REPZ (use only with string 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).
- 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
  - 66H—Operand-size override prefix.
- Group 4
  - 67H—Address-size override prefix.

For each instruction, one prefix may be used from each of these groups and be placed in any order. Using redundant prefixes (more than one prefix from a group) is reserved and may cause unpredictable behavior.

The LOCK prefix forces an atomic operation to insure exclusive use of shared memory in a multiprocessor environment. See “LOCK—Assert LOCK# Signal Prefix” in Chapter 3, *Instruction Set Reference*, for a detailed description of this prefix and the instructions with which it can be used.

The repeat prefixes cause an instruction to be repeated for each element of a string. They can be used only with the string instructions: MOVSB, CMPSB, SCASB, LODSB, and STOSB. Use of the repeat prefixes with other IA-32 instructions is reserved and may cause unpredictable behavior (see the note below).

The branch hint prefixes allow a program to give a hint to the processor about the most likely code path that will be taken at a branch. These prefixes can only be used with the conditional branch instructions (*Jcc*). Use of these prefixes with other IA-32 instructions is reserved and may cause unpredictable behavior. The branch hint prefixes were introduced in the Pentium 4 processors as part of the SSE2 extensions.

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either operand size can be the default. This prefix selects the non-default size. Use of this prefix with MMX, SSE, and/or SSE2 instructions is reserved and may cause unpredictable behavior (see the note below).

The address-size override prefix allows a program to switch between 16- and 32-bit addressing. Either address size can be the default. This prefix selects the non-default size. Using this prefix when the operands for an instruction do not reside in memory is reserved and may cause unpredictable behavior.



### NOTE

Some of the SSE and SSE2 instructions have three-byte opcodes. For these three-byte opcodes, the third opcode byte may be F2H, F3H, or 66H. For example, the SSE2 instruction CVTDQ2PD has the three-byte opcode F3 OF E6. The third opcode byte of these three-byte opcodes should not be thought of as a prefix, even though it has the same encoding as the operand size prefix (66H) or one of the repeat prefixes (F2H and F3H). As described above, using the operand size and repeat prefixes with SSE and SSE2 instructions is reserved.

## 2.3. OPCODE

The primary opcode is either 1 or 2 bytes. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller encoding fields can be defined within the primary opcode. These fields define the direction of the operation, the size of displacements, the register encoding, condition codes, or sign extension. The encoding of fields in the opcode varies, depending on the class of operation.

## 2.4. MODR/M AND SIB BYTES

Most 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 can be combined with the *mod* field to encode an addressing mode.

Certain encodings of the ModR/M byte require a second addressing byte, the SIB byte, to fully specify the addressing form. 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 2.6., “Addressing-Mode Encoding of ModR/M and SIB Bytes”, for the encodings of the ModR/M and SIB bytes.

## 2.5. DISPLACEMENT AND IMMEDIATE BYTES

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

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

## 2.6. ADDRESSING-MODE ENCODING OF MODR/M AND SIB BYTES

The values and the corresponding addressing forms of the ModR/M and SIB bytes are shown in Tables 2-1 through 2-3. The 16-bit addressing forms specified by the ModR/M byte are in Table 2-1, and the 32-bit addressing forms specified by the ModR/M byte are in Table 2-2. Table 2-3 shows the 32-bit addressing forms specified by the SIB byte.

In Tables 2-1 and 2-2, the first column (labeled “Effective Address”) lists 32 different effective addresses that can be assigned to one operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 effective addresses give the different ways of specifying a memory location; the last eight (specified by the Mod field encoding 11B) give the ways of specifying the general-purpose, MMX, and XMM registers. Each of the register encodings list four possible registers. For example, the first register-encoding (selected by the R/M field encoding of 000B) indicates the general-purpose registers EAX, AX or AL, MMX register MM0, or XMM register XMM0. Which of these five registers is used is determined by the opcode byte and the operand-size attribute, which select either the EAX register (32 bits) or AX register (16 bits).

The second and third columns in Tables 2-1 and 2-2 gives the binary encodings of the Mod and R/M fields in the ModR/M byte, respectively, required to obtain the associated effective address listed in the first column. All 32 possible combinations of the Mod and R/M fields are listed.

Across the top of Tables 2-1 and 2-2, the eight possible values of the 3-bit Reg/Opcode field are listed, in decimal (sixth row from top) and in binary (seventh row from top). The seventh row is labeled “REG=”, which represents the use of these 3 bits to give the location of a second operand, which must be a general-purpose, MMX, or XMM register. If the instruction does not require a second operand to be specified, then the 3 bits of the Reg/Opcode field may be used as an extension of the opcode, which is represented by the sixth row, labeled “/digit (Opcode)”. The five rows above give the byte, word, and doubleword general-purpose registers, the MMX registers, and the XMM registers that correspond to the register numbers, with the same assignments as for the R/M field when Mod field encoding is 11B. As with the R/M field register options, which of the five possible registers is used is determined by the opcode byte along with the operand-size attribute.

The body of Tables 2-1 and 2-2 (under the label “Value of ModR/M Byte (in Hexadecimal)”) contains a 32 by 8 array giving all of the 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, and the row specifies bits 0, 1 and 2, and also bits 6 and 7.

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

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =	AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP <sup>1</sup> EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111		
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16 <sup>2</sup> [BX]	00	000 001 010 011 100 101 110 111	00 01 02 0B 0C 0D 0E 0F	08 09 0A 13 14 15 16 17	10 11 12 1B 1C 1D 1E 1F	18 19 1A 23 24 25 26 27	20 21 2A 2B 2C 2D 2E 2F	28 29 2A 2B 2C 2D 2E 2F	30 31 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 4A 4B 4C 4D 4E 4F	48 49 4A 53 54 55 56 57	50 51 5A 5B 5C 5D 5E 5F	60 61 6A 6B 6C 6D 6E 6F	68 69 6A 6B 6C 6D 6E 6F	70 71 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 8A 8B 8C 8D 8E 8F	88 89 8A 93 94 95 96 97	90 91 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		
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	

**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 following the ModR/M byte, to be added to the index.
3. The "disp8" nomenclature denotes an 8-bit displacement following the ModR/M byte, to be sign-extended and added to the index.

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

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	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
disp8[EAX] <sup>3</sup>	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EBX];		011	43	4B	53	5B	63	6B	73	7B
disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
disp32[--][--]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		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 following the SIB byte, to be added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index.

Table 2-3 is organized similarly to Tables 2-1 and 2-2, except that its body gives the 256 possible values of the SIB byte, in hexadecimal. Which of the 8 general-purpose registers will be used as base is indicated across the top of the table, along with the corresponding values of the base field (bits 0, 1 and 2) in decimal and binary. The rows indicate which register is used as the index (determined by bits 3, 4 and 5) along with the scaling factor (determined by bits 6 and 7).

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

r32 Base = 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

**NOTE:**

1. The [\*] nomenclature means a disp32 with no base if MOD is 00, [EBP] otherwise. This provides the following addressing modes:

disp32[index] (MOD=00).  
 disp8[EBP][index](MOD=01).  
 disp32[EBP][index](MOD=10).

# 3

## **Instruction Set Reference**







# CHAPTER 3

## INSTRUCTION SET REFERENCE

This chapter describes the complete IA-32 instruction set, including the general-purpose, x87 FPU, MMX, SSE, SSE2, and system instructions. The instruction descriptions are arranged in alphabetical order. For each instruction, the forms are given for each operand combination, including the opcode, operands required, and a description. Also given for each instruction are a description of the instruction and its operands, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of the exceptions that can be generated.

### 3.1. INTERPRETING THE INSTRUCTION REFERENCE PAGES

This section describes the information contained in the various sections of the instruction reference pages that make up the majority of this chapter. It also explains the notational conventions and abbreviations used in these sections.

#### 3.1.1. Instruction Format

The following is an example of the format used for each IA-32 instruction description in this chapter:

##### CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement carry flag

##### 3.1.1.1. OPCODE COLUMN

The “Opcode” column gives the complete object code produced for each form of the instruction. When possible, the 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:

- **/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 both a register operand and an r/m operand.

- **cb, cw, cd, cp**—A 1-byte (cb), 2-byte (cw), 4-byte (cd), or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id**—A 1-byte (ib), 2-byte (iw), or 4-byte (id) 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 and doublewords are given with the low-order byte first.
- **+rb, +rw, +rd**—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. The register codes are given in Table 3-3.
- **+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 Encodings Associated with the +rb, +rw, and +rd Nomenclature**

rb			rw			rd		
AL	=	0	AX	=	0	EAX	=	0
CL	=	1	CX	=	1	ECX	=	1
DL	=	2	DX	=	2	EDX	=	2
BL	=	3	BX	=	3	EBX	=	3
rb			rw			rd		
AH	=	4	SP	=	4	ESP	=	4
CH	=	5	BP	=	5	EBP	=	5
DH	=	6	SI	=	6	ESI	=	6
BH	=	7	DI	=	7	EDI	=	7

### 3.1.1.2. INSTRUCTION COLUMN

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 and 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 and ptr16:32**—A far pointer, typically in 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, or BH.
- **r16**—One of the word general-purpose registers AX, CX, DX, BX, SP, BP, SI, or DI.
- **r32**—One of the doubleword general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.
- **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.
- **r/m8**—A byte operand that is either the contents of a byte general-purpose register (AL, BL, CL, DL, AH, BH, CH, and DH), or a byte from memory.
- **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, BX, CX, DX, SP, BP, SI, and DI. The contents of memory are found at the address provided by the effective address computation.
- **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, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. The contents of memory are found at the address provided by the effective address computation.
- **m**—A 16- or 32-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. This nomenclature is used only with the string instructions and the XLAT instruction.
- **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. This nomenclature is used only with the CMPXCHG8B instruction.
- **m128**—A memory double quadword operand in memory. This nomenclature is used only with the SSE and SSE2 instructions.

- **m16:16, m16:32**—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**—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.
- **moffs8, moffs16, moffs32**—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*<sup>th</sup> element from the top of the FPU register stack. (*i* ← 0 through 7)
- **mm**—An MMX register. The 64-bit MMX registers are: MM0 through MM7.
- **mm/m32**—The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **mm/m64**—An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **xmm**—An XMM register. The 128-bit XMM registers are: XMM0 through XMM7.
- **xmm/m32**—An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7. 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. 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. The contents of memory are found at the address provided by the effective address computation.

### 3.1.1.3. DESCRIPTION COLUMN

The “Description” column following the “Instruction” column briefly explains the various forms of the instruction. The following “Description” and “Operation” sections contain more details of the instruction's operation.

### 3.1.1.4. DESCRIPTION

The “Description” section describes the purpose of the instructions and the required operands. It also discusses the effect of the instruction on flags.

## 3.1.2. Operation

The “Operation” section contains an algorithmic description (written in pseudo-code) of the instruction. The pseudo-code uses a notation similar to the Algol or Pascal language. The 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 and ESAC 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 overridden segment.
- Parentheses around the “E” in a general-purpose register name, such as (E)SI, indicates that an offset is read from the SI register if the current address-size attribute is 16 or is read from the ESI register if the address-size attribute is 32.
- Brackets are also 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 contents 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, respectively, 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 either 16 or 32 bits. The **AddressSize** identifier represents the address-size attribute, which is either 16 or 32 bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the CMPS instruction used.

```
IF instruction ← CMPSW
  THEN OperandSize ← 16;
  ELSE
    IF instruction ← CMPSD
      THEN OperandSize ← 32;
    FI;
FI;
```

See “Operand-Size and Address-Size Attributes” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for general guidelines on how these attributes are determined.

- **StackAddrSize**—Represents the stack address-size attribute associated with the instruction, which has a value of 16 or 32 bits (see “Address-Size Attribute for Stack” in Chapter 6 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*).
- **SRC**—Represents the source operand.
- **DEST**—Represents the destination operand.

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  $00000F6H$ . 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” section in “PUSH—Push Word or Doubleword Onto the Stack” in this chapter for more information on the push operation.
- **Pop()** removes the value from the top of the stack and returns it. The statement  $EAX \leftarrow \text{Pop}()$ ; assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute. See the “Operation” section in Chapter 3, “POP—Pop a Value from the Stack” for more information on the pop operation.
- **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, which 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 base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register. An example, the function  $\text{Bit}[EAX, 21]$  is illustrated in Figure 3-1.

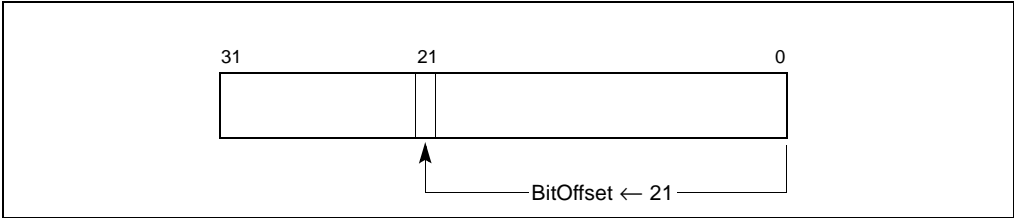


Figure 3-1. Bit Offset for BIT[EAX,21]

If BitBase is a memory address, BitOffset can range from -2 GBits to 2 GBits. 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. This operation is illustrated in Figure 3-2.

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

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 executables runs 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. Please refer to the *Intel C/C++ Compiler User's Guide With Support for the Streaming SIMD Extensions 2* (Order Number 718195-2001). See Appendix C, *Intel C/C++ Compiler Intrinsics and Functional Equivalents* for more information on using intrinsics.

#### 3.1.3.1. THE INTRINSICS API

The benefit of coding with MMX technology intrinsics and the SSE and SSE2 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.

#### 3.1.3.2. MMX TECHNOLOGY INTRINSICS

The MMX technology intrinsics are based on a new `__m64` data type to represent 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 guide and the *Intel C/C++ Compiler User’s Guide With Support for the Streaming SIMD Extensions 2* (Order Number 718195-2001). Refer to Appendix C, *Intel C/C++ Compiler Intrinsics and Functional Equivalents* for more information on using intrinsics.

### 3.1.3.3. SSE AND SSE2 INTRINSICS

The SSE and SSE2 intrinsics all make use of the XMM registers of the Pentium III and Pentium 4 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, you can use the `declspec` statement as described in the *Intel C/C++ Compiler User’s Guide With Support for the Streaming SIMD Extensions 2* (Order Number 718195-2001).

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. Refer to Appendix C, *Intel C/C++ Compiler Intrinsics and Functional Equivalents* 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, and MMX technology when writing programs with the intrinsics.

Keep the following three important issues in mind:

- Certain intrinsics, such as `_mm_loadr_ps` and `_mm_cmpgt_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 will 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 the *Intel C/C++ Compiler User's Guide With Support for the Streaming SIMD Extensions 2* (Order Number 718195-2001). Refer to Appendix C, *Intel C/C++ Compiler Intrinsics and Functional Equivalents* for more information on using intrinsics.

### 3.1.4. Flags Affected

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 *IA-32 Intel Architecture 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.

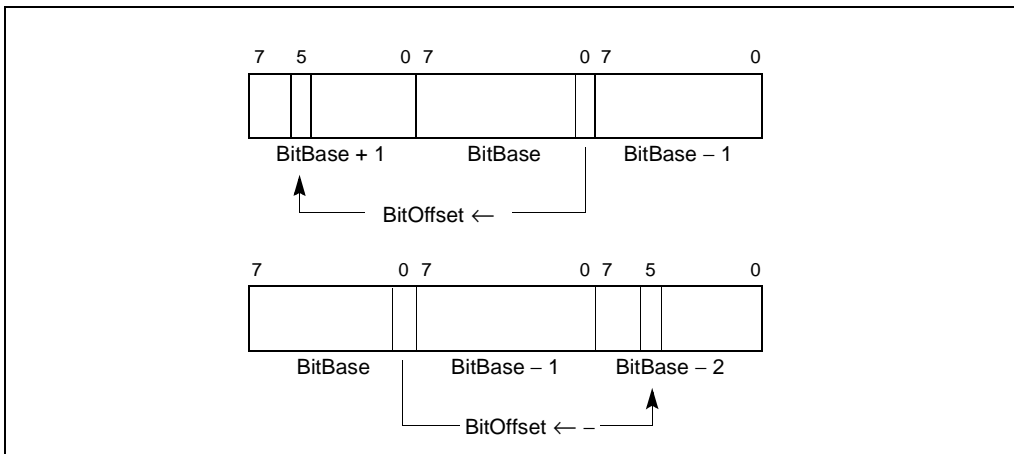


Figure 3-2. Memory Bit Indexing

### 3.1.5. FPU Flags Affected

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.

### 3.1.6. Protected Mode Exceptions

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 3-2 associates each two-letter mnemonic with the corresponding interrupt vector number and exception name. See Chapter 5, *Interrupt and Exception Handling*, in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, 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 3-2. IA-32 General Exceptions

Vector No.	Name	Source	Protected Mode	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
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*	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	#XF—SIMD Floating-Point Numeric Error	SSE and SSE2 floating-point instructions.	Yes	Yes	Yes

**NOTE:**

\* In the real-address mode, vector 13 is the segment overrun exception.

### 3.1.7. Real-Address Mode Exceptions

The “Real-Address Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-2).

### 3.1.8. Virtual-8086 Mode Exceptions

The “Virtual-8086 Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-2).

### 3.1.9. Floating-Point Exceptions

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 3-3 associates a one- or two-letter mnemonic with the corresponding exception name. See “Floating-Point Exception Conditions” in Chapter 8 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a detailed description of these exceptions.

**Table 3-3. 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)

### 3.1.10. SIMD Floating-Point Exceptions

The “SIMD Floating-Point Exceptions” section lists exceptions that can occur when an SSE and SSE2 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XF, vector number 19) being generated. Table 3-4 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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*.

**Table 3-4. SIMD Floating-Point Exceptions**

<b>Mnemonic</b>	<b>Name</b>	<b>Source</b>
#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)

## 3.2. INSTRUCTION REFERENCE

The remainder of this chapter provides detailed descriptions of each of the IA-32 instructions.

## AAA—ASCII Adjust After Addition

Opcode	Instruction	Description
37	AAA	ASCII adjust AL after addition

### 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 is incremented 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 cleared to 0.

### Operation

IF ((AL AND 0FH) > 9) OR (AF ← 1)

THEN

AL ← (AL + 6);

AH ← AH + 1;

AF ← 1;

CF ← 1;

ELSE

AF ← 0;

CF ← 0;

FI;

AL ← AL AND 0FH;

### Flags Affected

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

### Exceptions (All Operating Modes)

None.

## AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Description
D5 0A	AAD	ASCII adjust AX before division
D5 <i>ib</i>	(No mnemonic)	Adjust AX before division to number base <i>imm8</i>

### 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*).

### Operation

```
tempAL ← AL;
tempAH ← AH;
AL ← (tempAL + (tempAH * imm8)) AND FFH; (* imm8 is set to 0AH for the AAD mnemonic *)
AH ← 0
```

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.

### Exceptions (All Operating Modes)

None.



## AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Description
D4 0A	AAM	ASCII adjust AX after multiply
D4 <i>ib</i>	(No mnemonic)	Adjust AX after multiply to number base <i>imm8</i>

### 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*).

### Operation

```
tempAL ← AL;
AH ← tempAL / imm8; (* imm8 is set to 0AH for the AAD mnemonic *)
AL ← tempAL MOD imm8;
```

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.

### Exceptions (All Operating Modes)

None with the default immediate value of 0AH. If, however, an immediate value of 0 is used, it will cause a #DE (divide error) exception.

## AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Description
3F	AAS	ASCII adjust AL after subtraction

### 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 is decremented 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 nibble set to 0.

### Operation

IF ((AL AND 0FH) > 9) OR (AF ← 1)

THEN

AL ← AL – 6;

AH ← AH – 1;

AF ← 1;

CF ← 1;

ELSE

CF ← 0;

AF ← 0;

FI;

AL ← AL AND 0FH;

### 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.

### Exceptions (All Operating Modes)

None.

## ADC—Add with Carry

Opcode	Instruction	Description
14 <i>ib</i>	ADC AL, <i>imm8</i>	Add with carry <i>imm8</i> to AL
15 <i>iw</i>	ADC AX, <i>imm16</i>	Add with carry <i>imm16</i> to AX
15 <i>id</i>	ADC EAX, <i>imm32</i>	Add with carry <i>imm32</i> to EAX
80 /2 <i>ib</i>	ADC <i>r/m8</i> , <i>imm8</i>	Add with carry <i>imm8</i> to <i>r/m8</i>
81 /2 <i>iw</i>	ADC <i>r/m16</i> , <i>imm16</i>	Add with carry <i>imm16</i> to <i>r/m16</i>
81 /2 <i>id</i>	ADC <i>r/m32</i> , <i>imm32</i>	Add with CF <i>imm32</i> to <i>r/m32</i>
83 /2 <i>ib</i>	ADC <i>r/m16</i> , <i>imm8</i>	Add with CF sign-extended <i>imm8</i> to <i>r/m16</i>
83 /2 <i>ib</i>	ADC <i>r/m32</i> , <i>imm8</i>	Add with CF sign-extended <i>imm8</i> into <i>r/m32</i>
10 <i>lr</i>	ADC <i>r/m8</i> , <i>r8</i>	Add with carry byte register to <i>r/m8</i>
11 <i>lr</i>	ADC <i>r/m16</i> , <i>r16</i>	Add with carry <i>r16</i> to <i>r/m16</i>
11 <i>lr</i>	ADC <i>r/m32</i> , <i>r32</i>	Add with CF <i>r32</i> to <i>r/m32</i>
12 <i>lr</i>	ADC <i>r8</i> , <i>r/m8</i>	Add with carry <i>r/m8</i> to byte register
13 <i>lr</i>	ADC <i>r16</i> , <i>r/m16</i>	Add with carry <i>r/m16</i> to <i>r16</i>
13 <i>lr</i>	ADC <i>r32</i> , <i>r/m32</i>	Add with CF <i>r/m32</i> to <i>r32</i>

### 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.

### Operation

DEST ← DEST + SRC + CF;

### Flags Affected

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

## ADC—Add with Carry (Continued)

### 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.

### 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.

## ADD—Add

Opcode	Instruction	Description
04 <i>ib</i>	ADD AL, <i>imm8</i>	Add <i>imm8</i> to AL
05 <i>iw</i>	ADD AX, <i>imm16</i>	Add <i>imm16</i> to AX
05 <i>id</i>	ADD EAX, <i>imm32</i>	Add <i>imm32</i> to EAX
80 /0 <i>ib</i>	ADD <i>r/m8</i> , <i>imm8</i>	Add <i>imm8</i> to <i>r/m8</i>
81 /0 <i>iw</i>	ADD <i>r/m16</i> , <i>imm16</i>	Add <i>imm16</i> to <i>r/m16</i>
81 /0 <i>id</i>	ADD <i>r/m32</i> , <i>imm32</i>	Add <i>imm32</i> to <i>r/m32</i>
83 /0 <i>ib</i>	ADD <i>r/m16</i> , <i>imm8</i>	Add sign-extended <i>imm8</i> to <i>r/m16</i>
83 /0 <i>ib</i>	ADD <i>r/m32</i> , <i>imm8</i>	Add sign-extended <i>imm8</i> to <i>r/m32</i>
00 /r	ADD <i>r/m8</i> , <i>r8</i>	Add <i>r8</i> to <i>r/m8</i>
01 /r	ADD <i>r/m16</i> , <i>r16</i>	Add <i>r16</i> to <i>r/m16</i>
01 /r	ADD <i>r/m32</i> , <i>r32</i>	Add <i>r32</i> to <i>r/m32</i>
02 /r	ADD <i>r8</i> , <i>r/m8</i>	Add <i>r/m8</i> to <i>r8</i>
03 /r	ADD <i>r16</i> , <i>r/m16</i>	Add <i>r/m16</i> to <i>r16</i>
03 /r	ADD <i>r32</i> , <i>r/m32</i>	Add <i>r/m32</i> to <i>r32</i>

### Description

Adds the first operand (destination operand) and the second operand (source 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, 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.

### Operation

DEST ← DEST + SRC;

### Flags Affected

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

## ADD—Add (Continued)

### 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.

### 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.

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

Opcode	Instruction	Description
66 0F 58 /r	ADDPD <i>xmm1</i> , <i>xmm2/m128</i>	Add packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .

### 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. 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD double-precision floating-point operation.

### Operation

DEST[63-0] ← DEST[63-0] + SRC[63-0];  
 DEST[127-64] ← DEST[127-64] + SRC[127-64];

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

ADDPD            \_\_m128d \_mm\_add\_pd (m128d a, m128d b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.

## ADDPD—Add Packed Double-Precision Floating-Point Values (Continued)

If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault
-----------------	------------------



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

Opcode	Instruction	Description
0F 5B /r	ADDPS <i>xmm1</i> , <i>xmm2/m128</i>	Add packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .

### 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. 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD single-precision floating-point operation.

### Operation

```
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];
```

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

```
ADDPS    __m128 __mm_add_ps(__m128 a, __m128 b)
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

## ADDPS—Add Packed Single-Precision Floating-Point Values (Continued)

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

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

Opcode	Instruction	Description
F2 0F 58 /r	ADDSD <i>xmm1</i> , <i>xmm2/m64</i>	Add the low double-precision floating-point value from <i>xmm2/m64</i> to <i>xmm1</i> .

### 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. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

### Operation

DEST[63-0] ← DEST[63-0] + SRC[63-0];

\* DEST[127-64] remains unchanged \*;

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

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

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

## ADDSD—Add Scalar Double-Precision Floating-Point Values (Continued)

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

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

Opcode	Instruction	Description
F3 0F 58 /r	ADDSS <i>xmm1</i> , <i>xmm2/m32</i>	Add the low single-precision floating-point value from <i>xmm2/m32</i> to <i>xmm1</i> .

### 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. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

### Operation

DEST[31-0] ← DEST[31-0] + SRC[31-0];  
 \* DEST[127-32] remain unchanged \*;

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

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

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

## ADDSS—Add Scalar Single-Precision Floating-Point Values (Continued)

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## AND—Logical AND

<b>Opcode</b>	<b>Instruction</b>	<b>Description</b>
24 <i>ib</i>	AND AL, <i>imm8</i>	AL AND <i>imm8</i>
25 <i>iw</i>	AND AX, <i>imm16</i>	AX AND <i>imm16</i>
25 <i>id</i>	AND EAX, <i>imm32</i>	EAX AND <i>imm32</i>
80 /4 <i>ib</i>	AND <i>r/m8,imm8</i>	<i>r/m8</i> AND <i>imm8</i>
81 /4 <i>iw</i>	AND <i>r/m16,imm16</i>	<i>r/m16</i> AND <i>imm16</i>
81 /4 <i>id</i>	AND <i>r/m32,imm32</i>	<i>r/m32</i> AND <i>imm32</i>
83 /4 <i>ib</i>	AND <i>r/m16,imm8</i>	<i>r/m16</i> AND <i>imm8</i> (sign-extended)
83 /4 <i>ib</i>	AND <i>r/m32,imm8</i>	<i>r/m32</i> AND <i>imm8</i> (sign-extended)
20 /r	AND <i>r/m8,r8</i>	<i>r/m8</i> AND <i>r8</i>
21 /r	AND <i>r/m16,r16</i>	<i>r/m16</i> AND <i>r16</i>
21 /r	AND <i>r/m32,r32</i>	<i>r/m32</i> AND <i>r32</i>
22 /r	AND <i>r8,r/m8</i>	<i>r8</i> AND <i>r/m8</i>
23 /r	AND <i>r16,r/m16</i>	<i>r16</i> AND <i>r/m16</i>
23 /r	AND <i>r32,r/m32</i>	<i>r32</i> AND <i>r/m32</i>

### 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 instruction to be executed atomically.

### Operation

DEST ← DEST AND SRC;

### Flags Affected

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

### Protected Mode Exceptions

- #GP(0) If the destination operand points to a nonwritable 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.

**AND—Logical AND (Continued)**

#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**

#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.



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

Opcode	Instruction	Description
66 0F 54 /r	ANDPD <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise logical AND of <i>xmm2/m128</i> and <i>xmm1</i> .

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

### Operation

DEST[127:0] ← DEST[127:0] BitwiseAND SRC[127:0];

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

ANDPD            \_\_m128d \_mm\_and\_pd(\_\_m128d a, \_\_m128d b)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

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

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

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

Opcode	Instruction	Description
0F 54 /r	ANDPS <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise logical AND of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical AND 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.

### Operation

DEST[127:0] ← DEST[127:0] BitwiseAND SRC[127:0];

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

ANDPS            \_\_m128 \_mm\_and\_ps(\_\_m128 a, \_\_m128 b)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.

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

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

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

Opcode	Instruction	Description
66 0F 55 /r	ANDNPD <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise logical AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

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

### Operation

DEST[127:0] ← (NOT(DEST[127:0])) BitwiseAND (SRC[127:0]);

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

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

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.

## ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values (Continued)

If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

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

Opcode	Instruction	Description
0F 55 /r	ANDNPS <i>xmm1, xmm2/m128</i>	Bitwise logical AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .

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

### Operation

DEST[127:0] ← (NOT(DEST[127:0])) BitwiseAND (SRC[127:0]);

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

ANDNPS      `__m128 _mm_andnot_ps(__m128 a, __m128 b)`

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.

## ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values (Continued)

If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------



## ARPL—Adjust RPL Field of Segment Selector

Opcode	Instruction	Description
63 /r	ARPL r/m16,r16	Adjust RPL of r/m16 to not less than RPL of r16

### 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 insures 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.)

See “Checking Caller Access Privileges” in Chapter 4 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information about the use of this instruction.

### Operation

```
IF DEST[RPL] < SRC[RPL]
THEN
    ZF ← 1;
    DEST[RPL] ← SRC[RPL];
ELSE
    ZF ← 0;
FI;
```

### Flags Affected

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, is cleared to 0.

## ARPL—Adjust RPL Field of Segment Selector (Continued)

### Protected Mode Exceptions

- #GP(0)                    If the destination is located in a nonwritable 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                        The ARPL instruction is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

- #UD                        The ARPL instruction is not recognized in virtual-8086 mode.

## BOUND—Check Array Index Against Bounds

Opcode	Instruction	Description
62 /r	BOUND <i>r16</i> , <i>m16&amp;16</i>	Check if <i>r16</i> (array index) is within bounds specified by <i>m16&amp;16</i>
62 /r	BOUND <i>r32</i> , <i>m32&amp;32</i>	Check if <i>r32</i> (array index) is within bounds specified by <i>m16&amp;16</i>

### Description

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 a 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.

### Operation

```
IF (ArrayIndex < LowerBound OR ArrayIndex > UppderBound)
  (* Below lower bound or above upper bound *)
  THEN
    #BR;
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#BR	If the bounds test fails.
#UD	If second 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.
	If the DS, ES, FS, or GS register contains a null segment selector.

**BOUND—Check Array Index Against Bounds (Continued)**

#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.
#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.
#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.

## BSF—Bit Scan Forward

Opcode	Instruction	Description
0F BC	BSF <i>r16,r/m16</i>	Bit scan forward on <i>r/m16</i>
0F BC	BSF <i>r32,r/m32</i>	Bit scan forward on <i>r/m32</i>

### 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 contents source operand are 0, the contents of the destination operand is undefined.

### Operation

```

IF SRC ← 0
  THEN
    ZF ← 1;
    DEST is undefined;
  ELSE
    ZF ← 0;
    temp ← 0;
    WHILE Bit(SRC, temp) ← 0
      DO
        temp ← temp + 1;
        DEST ← temp;
      OD;
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.

## BSF—Bit Scan Forward (Continued)

### 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.               |

## BSR—Bit Scan Reverse

Opcode	Instruction	Description
OF BD	BSR <i>r16,r/m16</i>	Bit scan reverse on <i>r/m16</i>
OF BD	BSR <i>r32,r/m32</i>	Bit scan reverse on <i>r/m32</i>

### 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 contents source operand are 0, the contents of the destination operand is undefined.

### 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;
      DEST ← temp;
    OD;
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.

## BSR—Bit Scan Reverse (Continued)

### 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.               |



## BSWAP—Byte Swap

Opcode	Instruction	Description
0F C8+rd	BSWAP r32	Reverses the byte order of a 32-bit register.

### Description

Reverses the byte order of a 32-bit (destination) register: bits 0 through 7 are swapped with bits 24 through 31, and bits 8 through 15 are swapped with bits 16 through 23. 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.

### IA-32 Architecture Compatibility

The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486 processor family. For compatibility with this instruction, include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

### Operation

```
TEMP ← DEST
DEST[7..0] ← TEMP[31..24]
DEST[15..8] ← TEMP[23..16]
DEST[23..16] ← TEMP[15..8]
DEST[31..24] ← TEMP[7..0]
```

### Flags Affected

None.

### Exceptions (All Operating Modes)

None.

## BT—Bit Test

Opcode	Instruction	Description
0F A3	BT <i>r/m16,r16</i>	Store selected bit in CF flag
0F A3	BT <i>r/m32,r32</i>	Store selected bit in CF flag
0F BA /4 <i>ib</i>	BT <i>r/m16,imm8</i>	Store selected bit in CF flag
0F BA /4 <i>ib</i>	BT <i>r/m32,imm8</i>	Store selected bit in CF flag

### 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) 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 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range  $-2^{31}$  to  $2^{31} - 1$  for a register offset and 0 to 31 for an immediate offset.

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.

### Operation

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

## BT—Bit Test (Continued)

### Flags Affected

The CF flag contains the value of the selected bit. The OF, 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.

### 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.

## BTC—Bit Test and Complement

Opcode	Instruction	Description
0F BB	BTC <i>r/m16,r16</i>	Store selected bit in CF flag and complement
0F BB	BTC <i>r/m32,r32</i>	Store selected bit in CF flag and complement
0F BA /7 <i>ib</i>	BTC <i>r/m16,imm8</i>	Store selected bit in CF flag and complement
0F BA /7 <i>ib</i>	BTC <i>r/m32,imm8</i>	Store selected bit in CF flag and complement

### 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 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range  $-2^{31}$  to  $2^{31} - 1$  for a register offset and 0 to 31 for an immediate offset.

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.

### Operation

CF  $\leftarrow$  Bit(BitBase, BitOffset)  
 Bit(BitBase, BitOffset)  $\leftarrow$  NOT Bit(BitBase, BitOffset);

### Flags Affected

The CF flag contains the value of the selected bit before it is complemented. The OF, SF, ZF, AF, and PF flags are undefined.

### Protected Mode Exceptions

- #GP(0) If the destination operand points to a nonwritable 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.

## BTC—Bit Test and Complement (Continued)

#PF(fault-code)	If a 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.

### 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.

## BTR—Bit Test and Reset

Opcode	Instruction	Description
0F B3	BTR <i>r/m16,r16</i>	Store selected bit in CF flag and clear
0F B3	BTR <i>r/m32,r32</i>	Store selected bit in CF flag and clear
0F BA /6 <i>ib</i>	BTR <i>r/m16,imm8</i>	Store selected bit in CF flag and clear
0F BA /6 <i>ib</i>	BTR <i>r/m32,imm8</i>	Store selected bit in CF flag and clear

### 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 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range  $-2^{31}$  to  $2^{31} - 1$  for a register offset and 0 to 31 for an immediate offset.

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.

### 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 OF, SF, ZF, AF, and PF flags are undefined.

### Protected Mode Exceptions

- #GP(0)                    If the destination operand points to a nonwritable 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.

## BTR—Bit Test and Reset (Continued)

- #PF(fault-code)      If a 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.

### 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.

## BTS—Bit Test and Set

Opcode	Instruction	Description
0F AB	BTS <i>r/m16,r16</i>	Store selected bit in CF flag and set
0F AB	BTS <i>r/m32,r32</i>	Store selected bit in CF flag and set
0F BA /5 <i>ib</i>	BTS <i>r/m16,imm8</i>	Store selected bit in CF flag and set
0F BA /5 <i>ib</i>	BTS <i>r/m32,imm8</i>	Store selected bit in CF flag and set

### 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 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range  $-2^{31}$  to  $2^{31} - 1$  for a register offset and 0 to 31 for an immediate offset.

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.

### 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 OF, SF, ZF, AF, and PF flags are undefined.

### Protected Mode Exceptions

- #GP(0)                    If the destination operand points to a nonwritable 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.



## BTS—Bit Test and Set (Continued)

#PF(fault-code)	If a 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.

### 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.

## CALL—Call Procedure

Opcode	Instruction	Description
E8 <i>cw</i>	CALL <i>rel16</i>	Call near, relative, displacement relative to next instruction
E8 <i>cd</i>	CALL <i>rel32</i>	Call near, relative, displacement relative to next instruction
FF /2	CALL <i>r/m16</i>	Call near, absolute indirect, address given in <i>r/m16</i>
FF /2	CALL <i>r/m32</i>	Call near, absolute indirect, address given in <i>r/m32</i>
9A <i>cd</i>	CALL <i>ptr16:16</i>	Call far, absolute, address given in operand
9A <i>cp</i>	CALL <i>ptr16:32</i>	Call far, absolute, address given in operand
FF /3	CALL <i>m16:16</i>	Call far, absolute indirect, address given in <i>m16:16</i>
FF /3	CALL <i>m16:32</i>	Call far, absolute indirect, address given in <i>m16:32</i>

### Description

Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. 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 calls:

- Near call—A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call—A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment 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 the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 6, *Task Management*, in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, 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) onto the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified with the target operand. The target operand specifies either an absolute offset in the code segment (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, which points to the instruction following the CALL instruction). The CS register is not changed on near calls.

## CALL—Call Procedure (Continued)

For a near call, 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 to 0s, resulting in a maximum instruction pointer size of 16 bits. (When accessing an absolute offset indirectly using the stack pointer [ESP] as a 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 register. As with absolute offsets, the operand-size attribute determines the size of the target operand (16 or 32 bits).

**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 onto 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. 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 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 to 0s.

**Far Calls in Protected Mode.** When the processor is operating in protected mode, the CALL instruction can be used to perform the following three 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, and the offset from the instruction is loaded into the EIP register.

## CALL—Call Procedure (Continued)

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 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. Here again, 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 procedure's 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 somewhat similar to executing a call 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 in 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 EIP register so that the task begins executing again at this next instruction.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 6, *Task Management*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for detailed information on the mechanics of a task switch.

Note that when you execute at 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, will automatically use the previous task link to return to the calling task. (See "Task Linking" in Chapter 6 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from the JMP instruction which does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

## CALL—Call Procedure (Continued)

**Mixing 16-Bit and 32-Bit Calls.** When making far calls between 16-bit and 32-bit code segments, the calls should be made through 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 is saved. Also, the call should be made using a 16-bit call gate so that 16-bit values will be pushed on the stack. See Chapter 16, *Mixing 16-Bit and 32-Bit Code*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information on making calls between 16-bit and 32-bit code segments.

### Operation

```

IF near call
  THEN IF near relative call
    IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
    THEN IF OperandSize ← 32
      THEN
        IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
        Push(EIP);
        EIP ← EIP + DEST; (* DEST is rel32 *)
      ELSE (* OperandSize ← 16 *)
        IF stack not large enough for a 2-byte return address THEN #SS(0); FI;
        Push(IP);
        EIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
      FI;
    FI;
  ELSE (* near absolute call *)
    IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
    IF OperandSize ← 32
      THEN
        IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
        Push(EIP);
        EIP ← DEST; (* DEST is r/m32 *)
      ELSE (* OperandSize ← 16 *)
        IF stack not large enough for a 2-byte return address THEN #SS(0); FI;
        Push(IP);
        EIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
      FI;
    FI;
  FI;
IF far call AND (PE ← 0 OR (PE ← 1 AND VM ← 1)) (* real-address or virtual-8086 mode *)
  THEN
    IF OperandSize ← 32
      THEN
        IF stack not large enough for a 6-byte return address THEN #SS(0); FI;
        IF the instruction pointer is not within code segment limit THEN #GP(0); FI;

```

**CALL—Call Procedure (Continued)**

Push(CS); (\* padded with 16 high-order bits \*)

Push(EIP);

CS ← DEST[47:32]; (\* DEST is *ptr16:32* or [*m16:32*] \*)

EIP ← DEST[31:0]; (\* DEST is *ptr16:32* or [*m16:32*] \*)

ELSE (\* OperandSize ← 16 \*)

IF stack not large enough for a 4-byte return address THEN #SS(0); FI;

IF the instruction pointer is not within code segment limit THEN #GP(0); FI;

Push(CS);

Push(IP);

CS ← DEST[31:16]; (\* DEST is *ptr16:16* or [*m16:16*] \*)

EIP ← DEST[15:0]; (\* DEST is *ptr16:16* or [*m16:16*] \*)

EIP ← EIP AND 0000FFFFH; (\* clear upper 16 bits \*)

FI;

FI;

IF far call AND (PE ← 1 AND VM ← 0) (\* Protected mode, not virtual-8086 mode \*)

THEN

IF segment selector in target operand null THEN #GP(0); FI;

IF segment selector index not within descriptor table limits

THEN #GP(new code segment selector);

FI;

Read type and access rights of selected segment descriptor;

IF segment type is not a conforming or nonconforming code segment, call gate,  
task gate, or TSS THEN #GP(segment selector); FI;

Depending on type and access rights

GO TO CONFORMING-CODE-SEGMENT;

GO TO NONCONFORMING-CODE-SEGMENT;

GO TO CALL-GATE;

GO TO TASK-GATE;

GO TO TASK-STATE-SEGMENT;

FI;

CONFORMING-CODE-SEGMENT:

IF DPL > CPL THEN #GP(new code segment selector); FI;

IF segment not present THEN #NP(new code segment selector); FI;

IF OperandSize ← 32

THEN

IF stack not large enough for a 6-byte return address THEN #SS(0); FI;

IF the instruction pointer is not within code segment limit THEN #GP(0); FI;

Push(CS); (\* padded with 16 high-order bits \*)

Push(EIP);

CS ← DEST[NewCodeSegmentSelector];

(\* segment descriptor information also loaded \*)

CS(RPL) ← CPL

EIP ← DEST[offset];

**CALL—Call Procedure (Continued)**

```

ELSE (* OperandSize ← 16 *)
  IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
  IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
  Push(CS);
  Push(IP);
  CS ← DEST[NewCodeSegmentSelector];
  (* segment descriptor information also loaded *)
  CS(RPL) ← CPL
  EIP ← DEST[offset] AND 0000FFFFH; (* clear upper 16 bits *)

```

```

FI;

```

```

END;

```

**NONCONFORMING-CODE-SEGMENT:**

```

IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(new code segment selector); FI;
IF segment not present THEN #NP(new code segment selector); FI;
IF stack not large enough for return address THEN #SS(0); FI;
tempEIP ← DEST[offset]
IF OperandSize=16
  THEN
    tempEIP ← tempEIP AND 0000FFFFH; (* clear upper 16 bits *)

```

```

FI;

```

```

IF tempEIP outside code segment limit THEN #GP(0); FI;

```

```

IF OperandSize ← 32

```

```

  THEN
    Push(CS); (* padded with 16 high-order bits *)
    Push(EIP);
    CS ← DEST[NewCodeSegmentSelector];
    (* segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    EIP ← tempEIP;

```

```

  ELSE (* OperandSize ← 16 *)

```

```

    Push(CS);
    Push(IP);
    CS ← DEST[NewCodeSegmentSelector];
    (* segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    EIP ← tempEIP;

```

```

FI;

```

```

END;

```

**CALL-GATE:**

```

IF call gate DPL < CPL or RPL THEN #GP(call gate selector); FI;
IF call gate not present THEN #NP(call gate selector); FI;
IF call gate code-segment selector is null THEN #GP(0); FI;

```

**CALL—Call Procedure (Continued)**

```

IF call gate code-segment selector index is outside descriptor table limits
    THEN #GP(code segment selector); FI;
Read code segment descriptor;
IF code-segment segment descriptor does not indicate a code segment
OR code-segment segment descriptor DPL > CPL
    THEN #GP(code segment selector); FI;
IF code segment not present THEN #NP(new code segment selector); FI;
IF code segment is non-conforming AND DPL < CPL
    THEN go to MORE-PRIVILEGE;
    ELSE go to SAME-PRIVILEGE;
FI;
END;

MORE-PRIVILEGE:
IF current TSS is 32-bit TSS
    THEN
        TSSstackAddress ← new code segment (DPL * 8) + 4
        IF (TSSstackAddress + 7) > TSS limit
            THEN #TS(current TSS selector); FI;
        newSS ← TSSstackAddress + 4;
        newESP ← stack address;
    ELSE (* TSS is 16-bit *)
        TSSstackAddress ← new code segment (DPL * 4) + 2
        IF (TSSstackAddress + 4) > TSS limit
            THEN #TS(current TSS selector); FI;
        newESP ← TSSstackAddress;
        newSS ← TSSstackAddress + 2;
FI;
IF stack segment selector is null THEN #TS(stack segment selector); FI;
IF stack segment selector index is not within its descriptor table limits
    THEN #TS(SS selector); FI
Read code segment descriptor;
IF stack segment selector's RPL ≠ DPL of code segment
    OR stack segment DPL ≠ DPL of code segment
    OR stack segment is not a writable data segment
    THEN #TS(SS selector); FI
IF stack segment not present THEN #SS(SS selector); FI;
IF CallGateSize ← 32
    THEN
        IF stack does not have room for parameters plus 16 bytes
            THEN #SS(SS selector); FI;
        IF CallGate(InstructionPointer) not within code segment limit THEN #GP(0); FI;
        SS ← newSS;
        (* segment descriptor information also loaded *)

```



**CALL—Call Procedure (Continued)**

```

    ESP ← newESP;
    CS:EIP ← CallGate(CS:InstructionPointer);
    (* segment descriptor information also loaded *)
    Push(oldSS:oldESP); (* from calling procedure *)
    temp ← parameter count from call gate, masked to 5 bits;
    Push(parameters from calling procedure's stack, temp)
    Push(oldCS:oldEIP); (* return address to calling procedure *)
ELSE (* CallGateSize ← 16 *)
    IF stack does not have room for parameters plus 8 bytes
        THEN #SS(SS selector); FI;
    IF (CallGate(InstructionPointer) AND FFFFH) not within code segment limit
        THEN #GP(0); FI;
    SS ← newSS;
    (* segment descriptor information also loaded *)
    ESP ← newESP;
    CS:IP ← CallGate(CS:InstructionPointer);
    (* segment descriptor information also loaded *)
    Push(oldSS:oldESP); (* from calling procedure *)
    temp ← parameter count from call gate, masked to 5 bits;
    Push(parameters from calling procedure's stack, temp)
    Push(oldCS:oldEIP); (* return address to calling procedure *)

FI;
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); FI;
            IF EIP not within code segment limit then #GP(0); FI;
            CS:EIP ← CallGate(CS:EIP) (* segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* return address to calling procedure *)
        ELSE (* CallGateSize ← 16 *)
            IF stack does not have room for parameters plus 4 bytes
                THEN #SS(0); FI;
            IF IP not within code segment limit THEN #GP(0); FI;
            CS:IP ← CallGate(CS:instruction pointer)
            (* segment descriptor information also loaded *)
            Push(oldCS:oldIP); (* return address to calling procedure *)

    FI;
    CS(RPL) ← CPL
END;

```

**CALL—Call Procedure (Continued)**

## TASK-GATE:

IF task gate DPL < CPL or RPL  
 THEN #GP(task gate selector);

FI;

IF task gate not present  
 THEN #NP(task gate selector);

FI;

Read the TSS segment selector in the task-gate descriptor;

IF TSS segment selector local/global bit is set to local  
 OR index not within GDT limits  
 THEN #GP(TSS selector);

FI;

Access TSS descriptor in GDT;

IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)  
 THEN #GP(TSS selector);

FI;

IF TSS not present  
 THEN #NP(TSS selector);

FI;

SWITCH-TASKS (with nesting) to TSS;

IF EIP not within code segment limit  
 THEN #GP(0);

FI;

END;

## TASK-STATE-SEGMENT:

IF TSS DPL < CPL or RPL  
 OR TSS descriptor indicates TSS not available  
 THEN #GP(TSS selector);

FI;

IF TSS is not present  
 THEN #NP(TSS selector);

FI;

SWITCH-TASKS (with nesting) to TSS

IF EIP not within code segment limit  
 THEN #GP(0);

FI;

END;

**Flags Affected**

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

## CALL—Call Procedure (Continued)

### Protected Mode Exceptions

#GP(0)	<p>If target offset in destination operand is beyond the new code segment limit.</p> <p>If the segment selector in the destination operand is null.</p> <p>If the code segment selector in the gate is null.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</p>
#GP(selector)	<p>If code segment or gate or TSS selector index is outside descriptor table limits.</p> <p>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.</p> <p>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.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>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.</p> <p>If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.</p> <p>If the segment selector from a call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a call gate is greater than the CPL.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>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.</p> <p>If a memory operand effective address is outside the SS segment limit.</p>
#SS(selector)	<p>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.</p>

**CALL—Call Procedure (Continued)**

	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.

**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.
-----	---

**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.

## CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword

Opcode	Instruction	Description
98	CBW	AX ← sign-extend of AL
98	CWDE	EAX ← sign-extend of AX

### Description

Double the size of the source operand by means of sign extension (see Figure 7-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). 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 higher 16 bits of the EAX register.

The CBW and CWDE mnemonics reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16 and the CWDE instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CBW is used and to 32 when CWDE is used. Others may treat these mnemonics as synonyms (CBW/CWDE) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

The CWDE instruction is different from the CWD (convert word to double) instruction. The CWD instruction uses the DX:AX register pair as a destination operand; whereas, the CWDE instruction uses the EAX register as a destination.

### Operation

```
IF OperandSize ← 16 (* instruction ← CBW *)
  THEN AX ← SignExtend(AL);
  ELSE (* OperandSize ← 32, instruction ← CWDE *)
    EAX ← SignExtend(AX);
FI;
```

### Flags Affected

None.

### Exceptions (All Operating Modes)

None.

## **CDQ—Convert Double to Quad**

See entry for CWD/CDQ — Convert Word to Doubleword/Convert Doubleword to Quadword.

## CLC—Clear Carry Flag

Opcode	Instruction	Description
F8	CLC	Clear CF flag

### Description

Clears the CF flag in the EFLAGS register.

### Operation

$CF \leftarrow 0$ ;

### Flags Affected

The CF flag is cleared to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

### Exceptions (All Operating Modes)

None.

## CLD—Clear Direction Flag

Opcode	Instruction	Description
FC	CLD	Clear DF flag

### 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

$DF \leftarrow 0$ ;

### Flags Affected

The DF flag is cleared to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

### Exceptions (All Operating Modes)

None.



## CLFLUSH—Flush Cache Line

Opcode	Instruction	Description
0F AE /7	CLFLUSH <i>m8</i>	Flushes cache line containing <i>m8</i> .

### 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 the CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see Section , *CPUID—CPU Identification*). 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, 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 insure 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.

### Operation

Flush\_Cache\_Line(SRC)

## CLFLUSH—Cache Line Flush (Continued)

### 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 feature flag CLFSH is 0.

### Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID feature flag CLFSH is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CLI—Clear Interrupt Flag

Opcode	Instruction	Description
FA	CLI	Clear interrupt flag; interrupts disabled when interrupt flag cleared

### Description

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.

The following decision table indicates the action of the CLI instruction (bottom of the table) depending on the processor’s mode of operating and the CPL and IOPL of the currently running program or procedure (top of the table).

PE =	0	1	1	1	1
VM =	X	0	X	0	1
CPL	X	≤ IOPL	X	> IOPL	X
IOPL	X	X	= 3	X	< 3
IF ← 0	Y	Y	Y	N	N
#GP(0)	N	N	N	Y	Y

### NOTES:

X Don't care

N Action in column 1 not taken

Y Action in column 1 taken

### Operation

```

IF PE ← 0 (* Executing in real-address mode *)
  THEN
    IF ← 0;
  ELSE
    IF VM ← 0 (* Executing in protected mode *)
      THEN
        IF CPL ≤ IOPL
          THEN
            IF ← 0;
          ELSE
            #GP(0);
        FI;
      FI;
    FI;
  
```

**CLI—Clear Interrupt Flag (Continued)**

```

        ELSE (* Executing in Virtual-8086 mode *)
            IF IOPL ← 3
                THEN
                    IF ← 0
                ELSE
                    #GP(0);
            FI;
        FI;
FI;

```

**Flags Affected**

The IF is cleared 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.

**Protected Mode Exceptions**

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

**Real-Address Mode Exceptions**

None.

**Virtual-8086 Mode Exceptions**

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

## CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Description
0F 06	CLTS	Clears TS flag in CR0

### Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled “Control Registers” in Chapter 2 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, for more information about this flag.

### Operation

CR0(TS) ← 0;

### Flags Affected

The TS flag in CR0 register is cleared.

### Protected Mode Exceptions

#GP(0) If the CPL is greater than 0.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

#GP(0) If the CPL is greater than 0.

## CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement CF flag

### Description

Complements the CF flag in the EFLAGS register.

### Operation

CF ← NOT CF;

### 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)

None.

## CMOVcc—Conditional Move

<b>Opcode</b>	<b>Instruction</b>	<b>Description</b>
0F 47 /r	CMOVA <i>r16, r/m16</i>	Move if above (CF=0 and ZF=0)
0F 47 /r	CMOVA <i>r32, r/m32</i>	Move if above (CF=0 and ZF=0)
0F 43 /r	CMOVAE <i>r16, r/m16</i>	Move if above or equal (CF=0)
0F 43 /r	CMOVAE <i>r32, r/m32</i>	Move if above or equal (CF=0)
0F 42 /r	CMOVB <i>r16, r/m16</i>	Move if below (CF=1)
0F 42 /r	CMOVB <i>r32, r/m32</i>	Move if below (CF=1)
0F 46 /r	CMOVBE <i>r16, r/m16</i>	Move if below or equal (CF=1 or ZF=1)
0F 46 /r	CMOVBE <i>r32, r/m32</i>	Move if below or equal (CF=1 or ZF=1)
0F 42 /r	CMOVC <i>r16, r/m16</i>	Move if carry (CF=1)
0F 42 /r	CMOVC <i>r32, r/m32</i>	Move if carry (CF=1)
0F 44 /r	CMOVE <i>r16, r/m16</i>	Move if equal (ZF=1)
0F 44 /r	CMOVE <i>r32, r/m32</i>	Move if equal (ZF=1)
0F 4F /r	CMOVG <i>r16, r/m16</i>	Move if greater (ZF=0 and SF=OF)
0F 4F /r	CMOVG <i>r32, r/m32</i>	Move if greater (ZF=0 and SF=OF)
0F 4D /r	CMOVGE <i>r16, r/m16</i>	Move if greater or equal (SF=OF)
0F 4D /r	CMOVGE <i>r32, r/m32</i>	Move if greater or equal (SF=OF)
0F 4C /r	CMOVL <i>r16, r/m16</i>	Move if less (SF<>OF)
0F 4C /r	CMOVL <i>r32, r/m32</i>	Move if less (SF<>OF)
0F 4E /r	CMOVLE <i>r16, r/m16</i>	Move if less or equal (ZF=1 or SF<>OF)
0F 4E /r	CMOVLE <i>r32, r/m32</i>	Move if less or equal (ZF=1 or SF<>OF)
0F 46 /r	CMOVNA <i>r16, r/m16</i>	Move if not above (CF=1 or ZF=1)
0F 46 /r	CMOVNA <i>r32, r/m32</i>	Move if not above (CF=1 or ZF=1)
0F 42 /r	CMOVNAE <i>r16, r/m16</i>	Move if not above or equal (CF=1)
0F 42 /r	CMOVNAE <i>r32, r/m32</i>	Move if not above or equal (CF=1)
0F 43 /r	CMOVNB <i>r16, r/m16</i>	Move if not below (CF=0)
0F 43 /r	CMOVNB <i>r32, r/m32</i>	Move if not below (CF=0)
0F 47 /r	CMOVNBE <i>r16, r/m16</i>	Move if not below or equal (CF=0 and ZF=0)
0F 47 /r	CMOVNBE <i>r32, r/m32</i>	Move if not below or equal (CF=0 and ZF=0)
0F 43 /r	CMOVNC <i>r16, r/m16</i>	Move if not carry (CF=0)
0F 43 /r	CMOVNC <i>r32, r/m32</i>	Move if not carry (CF=0)
0F 45 /r	CMOVNE <i>r16, r/m16</i>	Move if not equal (ZF=0)
0F 45 /r	CMOVNE <i>r32, r/m32</i>	Move if not equal (ZF=0)
0F 4E /r	CMOVNG <i>r16, r/m16</i>	Move if not greater (ZF=1 or SF<>OF)
0F 4E /r	CMOVNG <i>r32, r/m32</i>	Move if not greater (ZF=1 or SF<>OF)
0F 4C /r	CMOVNGE <i>r16, r/m16</i>	Move if not greater or equal (SF<>OF)
0F 4C /r	CMOVNGE <i>r32, r/m32</i>	Move if not greater or equal (SF<>OF)
0F 4D /r	CMOVNL <i>r16, r/m16</i>	Move if not less (SF=OF)
0F 4D /r	CMOVNL <i>r32, r/m32</i>	Move if not less (SF=OF)
0F 4F /r	CMOVNLE <i>r16, r/m16</i>	Move if not less or equal (ZF=0 and SF=OF)
0F 4F /r	CMOVNLE <i>r32, r/m32</i>	Move if not less or equal (ZF=0 and SF=OF)

## CMOV $cc$ —Conditional Move (Continued)

Opcode	Instruction	Description
0F 41 /r	CMOVNO <i>r16, r/m16</i>	Move if not overflow (OF=0)
0F 41 /r	CMOVNO <i>r32, r/m32</i>	Move if not overflow (OF=0)
0F 4B /r	CMOVNP <i>r16, r/m16</i>	Move if not parity (PF=0)
0F 4B /r	CMOVNP <i>r32, r/m32</i>	Move if not parity (PF=0)
0F 49 /r	CMOVNS <i>r16, r/m16</i>	Move if not sign (SF=0)
0F 49 /r	CMOVNS <i>r32, r/m32</i>	Move if not sign (SF=0)
0F 45 /r	CMOVNZ <i>r16, r/m16</i>	Move if not zero (ZF=0)
0F 45 /r	CMOVNZ <i>r32, r/m32</i>	Move if not zero (ZF=0)
0F 40 /r	CMOVO <i>r16, r/m16</i>	Move if overflow (OF=0)
0F 40 /r	CMOVO <i>r32, r/m32</i>	Move if overflow (OF=0)
0F 4A /r	CMOVP <i>r16, r/m16</i>	Move if parity (PF=1)
0F 4A /r	CMOVP <i>r32, r/m32</i>	Move if parity (PF=1)
0F 4A /r	CMOVPE <i>r16, r/m16</i>	Move if parity even (PF=1)
0F 4A /r	CMOVPE <i>r32, r/m32</i>	Move if parity even (PF=1)
0F 4B /r	CMOVPO <i>r16, r/m16</i>	Move if parity odd (PF=0)
0F 4B /r	CMOVPO <i>r32, r/m32</i>	Move if parity odd (PF=0)
0F 48 /r	CMOVS <i>r16, r/m16</i>	Move if sign (SF=1)
0F 48 /r	CMOVS <i>r32, r/m32</i>	Move if sign (SF=1)
0F 44 /r	CMOVZ <i>r16, r/m16</i>	Move if zero (ZF=1)
0F 44 /r	CMOVZ <i>r32, r/m32</i>	Move if zero (ZF=1)

### Description

The CMOV $cc$  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 CMOV $cc$  instruction.

These instructions can move a 16- or 32-bit value 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 conditions for each CMOV $cc$  mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.



## CMOVcc—Conditional Move (Continued)

The CMOVcc instructions are new for the Pentium Pro processor family; however, they may not be supported by all the processors in the family. Software can determine if the CMOVcc 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).

### Operation

```
temp ← DEST
IF condition TRUE
    THEN
        DEST ← SRC
    ELSE
        DEST ← temp
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.

### 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.

## **CMOVcc—Conditional Move (Continued)**

#PF(fault-code)      If a page fault occurs.

#AC(0)                If alignment checking is enabled and an unaligned memory reference is made.

## CMP—Compare Two Operands

Opcode	Instruction	Description
3C <i>ib</i>	CMP AL, <i>imm8</i>	Compare <i>imm8</i> with AL
3D <i>iw</i>	CMP AX, <i>imm16</i>	Compare <i>imm16</i> with AX
3D <i>id</i>	CMP EAX, <i>imm32</i>	Compare <i>imm32</i> with EAX
80 /7 <i>ib</i>	CMP <i>r/m8</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m8</i>
81 /7 <i>iw</i>	CMP <i>r/m16</i> , <i>imm16</i>	Compare <i>imm16</i> with <i>r/m16</i>
81 /7 <i>id</i>	CMP <i>r/m32</i> , <i>imm32</i>	Compare <i>imm32</i> with <i>r/m32</i>
83 /7 <i>ib</i>	CMP <i>r/m16</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m16</i>
83 /7 <i>ib</i>	CMP <i>r/m32</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m32</i>
38 /r	CMP <i>r/m8</i> , <i>r8</i>	Compare <i>r8</i> with <i>r/m8</i>
39 /r	CMP <i>r/m16</i> , <i>r16</i>	Compare <i>r16</i> with <i>r/m16</i>
39 /r	CMP <i>r/m32</i> , <i>r32</i>	Compare <i>r32</i> with <i>r/m32</i>
3A /r	CMP <i>r8</i> , <i>r/m8</i>	Compare <i>r/m8</i> with <i>r8</i>
3B /r	CMP <i>r16</i> , <i>r/m16</i>	Compare <i>r/m16</i> with <i>r16</i>
3B /r	CMP <i>r32</i> , <i>r/m32</i>	Compare <i>r/m32</i> with <i>r32</i>

### 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 CMP instruction is typically used in conjunction with a conditional jump (*Jcc*), condition move (CMOV*cc*), or SET*cc* instruction. The condition codes used by the *Jcc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. Appendix B, *EFLAGS Condition Codes*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, shows the relationship of the status flags and the condition codes.

### 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.

## CMP—Compare Two Operands (Continued)

	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

#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.

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

Opcode	Instruction	Description
66 0F C2 /r ib	CMPPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	Compare packed double-precision floating-point values in <i>xmm2/m128</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.

### Description

Performs a SIMD compare of the two 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 source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The comparison predicate operand is an 8-bit immediate the first 3 bits of which define the type of comparison to be made (see Table 3-5); bits 4 through 7 of the immediate are reserved.

**Table 3-5. Comparison Predicate for CMPPD and CMPPS Instructions**

Predic- cate	imm8 Encod- ing	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emula- tion	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	$A \leq B$		False	Yes
		greater than	$A > B$	Swap Operands, Use LT	False	Yes
		greater-than-or- equal	$A \geq B$	Swap Operands, Use LE	False	Yes
UNORD	011B	unordered	$A, B = \text{Unordered}$		True	No
NEQ	100B	not-equal	$A \neq B$		True	No
NLT	101B	not-less-than	$\text{NOT}(A < B)$		True	Yes
NLE	110B	not-less-than-or- equal	$\text{NOT}(A \leq B)$		True	Yes
		not-greater-than	$\text{NOT}(A > B)$	Swap Operands, Use NLT	True	Yes
		not-greater-than- or-equal	$\text{NOT}(A \geq B)$	Swap Operands, Use NLE	True	Yes
ORD	111B	ordered	$A, B = \text{Ordered}$		False	No

## CMPPD—Compare Packed Double-Precision Floating-Point Values (Continued)

The unordered relationship is true when at least one of the two source operands being compared is a NaN or in an undefined format. The ordered relationship is true when neither source operand is a NaN or in an undefined format.

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 processor does 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 3-5 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction.

Pseudo-Op	CMPPD Implementation
CMPEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 0
CMPLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 1
CMPLEPD xmm1, xmm2	CMPPD xmm1, xmm2, 2
CMPUNORDPD xmm1, xmm2	CMPPD xmm1, xmm2, 3
CMPNEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 4
CMPNLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 5
CMPNLEPD xmm1, xmm2	CMPPD xmm1, xmm2, 6
CMPORDPD xmm1, xmm2	CMPPD xmm1, xmm2, 7

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

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP ← EQ;
- 1: OP ← LT;
- 2: OP ← LE;
- 3: OP ← UNORD;

## CMPPD—Compare Packed Double-Precision Floating-Point Values (Continued)

```

4: OP ← NEQ;
5: OP ← NLT;
6: OP ← NLE;
7: OP ← ORD;
DEFAULT:      Reserved;
CMP0 ← DEST[63-0] OP SRC[63-0];
CMP1 ← DEST[127-64] OP SRC[127-64];
IF CMP0 == TRUE
    THEN DEST[63-0] ← FFFFFFFFFFFFFFFFH
    ELSE DEST[63-0] ← 0000000000000000H; FI;
IF CMP1 == TRUE
    THEN DEST[127-64] ← FFFFFFFFFFFFFFFFH
    ELSE DEST[127-64] ← 0000000000000000H; FI;

```

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

CMPPD for equality	<code>__m128d _mm_cmpeq_pd(__m128d a, __m128d b)</code>
CMPPD for less-than	<code>__m128d _mm_cmplt_pd(__m128d a, __m128d b)</code>
CMPPD for less-than-or-equal	<code>__m128d _mm_cmple_pd(__m128d a, __m128d b)</code>
CMPPD for greater-than	<code>__m128d _mm_cmpgt_pd(__m128d a, __m128d b)</code>
CMPPD for greater-than-or-equal	<code>__m128d _mm_cmpge_pd(__m128d a, __m128d b)</code>
CMPPD for inequality	<code>__m128d _mm_cmpneq_pd(__m128d a, __m128d b)</code>
CMPPD for not-less-than	<code>__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)</code>
CMPPD for not-greater-than	<code>__m128d _mm_cmpngt_pd(__m128d a, __m128d b)</code>
CMPPD for not-greater-than-or-equal	<code>__m128d _mm_cmpnge_pd(__m128d a, __m128d b)</code>
CMPPD for ordered	<code>__m128d _mm_cmpord_pd(__m128d a, __m128d b)</code>
CMPPD for unordered	<code>__m128d _mm_cmpunord_pd(__m128d a, __m128d b)</code>
CMPPD for not-less-than-or-equal	<code>__m128d _mm_cmpnle_pd(__m128d a, __m128d b)</code>

### SIMD Floating-Point Exceptions

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

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.

## CMPPD—Compare Packed Double-Precision Floating-Point Values (Continued)

#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------



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

Opcode	Instruction	Description
0F C2 /r ib	CMPPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	Compare packed single-precision floating-point values in <i>xmm2/mem</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.

### Description

Performs a SIMD compare of the four 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 source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The comparison predicate operand is an 8-bit immediate the first 3 bits of which define the type of comparison to be made (see Table 3-5); bits 4 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN or in an undefined format. The ordered relationship is true when neither source operand is a NaN or in an undefined format.

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.

Some of the comparisons listed in Table 3-5 (such as the greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations) can be made only through software emulation. For these comparisons 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 3-5 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction:

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

## CMPPS—Compare Packed Single-Precision Floating-Point Values (Continued)

The greater-than relations not implemented by 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.)

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP ← EQ;
- 1: OP ← LT;
- 2: OP ← LE;
- 3: OP ← UNORD;
- 4: OP ← NE;
- 5: OP ← NLT;
- 6: OP ← NLE;
- 7: OP ← ORD;

EASC

```

CMP0 ← DEST[31-0] OP SRC[31-0];
CMP1 ← DEST[63-32] OP SRC[63-32];
CMP2 ← DEST [95-64] OP SRC[95-64];
CMP3 ← DEST[127-96] OP SRC[127-96];
IF CMP0 == TRUE
    THEN DEST[31-0] ← FFFFFFFFH
    ELSE DEST[31-0] ← 00000000H; FI;
IF CMP1 == TRUE
    THEN DEST[63-32] ← FFFFFFFFH
    ELSE DEST[63-32] ← 00000000H; FI;
IF CMP2 == TRUE
    THEN DEST[95-64] ← FFFFFFFFH
    ELSE DEST[95-64] ← 00000000H; FI;
IF CMP3 == TRUE
    THEN DEST[127-96] ← FFFFFFFFH
    ELSE DEST[127-96] ← 00000000H; FI;

```

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

CMPPS for equality	<code>__m128 _mm_cmpeq_ps(__m128 a, __m128 b)</code>
CMPPS for less-than	<code>__m128 _mm_cmplt_ps(__m128 a, __m128 b)</code>
CMPPS for less-than-or-equal	<code>__m128 _mm_cmple_ps(__m128 a, __m128 b)</code>
CMPPS for greater-than	<code>__m128 _mm_cmpgt_ps(__m128 a, __m128 b)</code>
CMPPS for greater-than-or-equal	<code>__m128 _mm_cmpge_ps(__m128 a, __m128 b)</code>
CMPPS for inequality	<code>__m128 _mm_cmpneq_ps(__m128 a, __m128 b)</code>

## CMPPS—Compare Packed Single-Precision Floating-Point Values (Continued)

CMPPS for not-less-than	<code>__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)</code>
CMPPS for not-greater-than	<code>__m128 _mm_cmpngt_ps(__m128 a, __m128 b)</code>
CMPPS for not-greater-than-or-equal	<code>__m128 _mm_cmpnge_ps(__m128 a, __m128 b)</code>
CMPPS for ordered	<code>__m128 _mm_cmpord_ps(__m128 a, __m128 b)</code>
CMPPS for unordered	<code>__m128 _mm_cmpunord_ps(__m128 a, __m128 b)</code>
CMPPS for not-less-than-or-equal	<code>__m128 _mm_cmpnle_ps(__m128 a, __m128 b)</code>

### SIMD Floating-Point Exceptions

Invalid, if SNaN operands, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.

## CMPPS—Compare Packed Single-Precision Floating-Point Values (Continued)

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CMPS/CMPSB/CMPSW/CMPD—Compare String Operands

Opcode	Instruction	Description
A6	CMPS m8, m8	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPS m16, m16	Compares word at address DS:(E)SI with word at address ES:(E)DI and sets the status flags accordingly
A7	CMPS m32, m32	Compares doubleword at address DS:(E)SI with doubleword at address ES:(E)DI and sets the status flags accordingly
A6	CMPSB	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPSW	Compares word at address DS:(E)SI with word at address ES:(E)DI and sets the status flags accordingly
A7	CMPD	Compares doubleword at address DS:(E)SI with doubleword at address ES:(E)DI and sets the status flags accordingly

### Description

Compares the byte, word, or double word specified with the first source operand with the byte, word, or double word specified with the second source operand and sets the status flags in the EFLAGS register according to the results. Both the source operands are located in memory. The address of the first source operand is read from either 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 second source operand is read from either 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 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-operands 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), but they do not have to specify the correct **location**. The locations of the source operands are always specified by the DS:(E)SI and ES:(E)DI 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 and ES:(E)DI 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), or CMPD (doubleword comparison).

## CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands (Continued)

After the comparison, 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 incremented; 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.

The CMPS, CMPSB, CMPSW, and CMPSD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. 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 this chapter for a description of the REP prefix.

### Operation

```
temp ← SRC1 – SRC2;
setStatusFlags(temp);
IF (byte comparison)
  THEN IF DF ← 0
    THEN
      (E)SI ← (E)SI + 1;
      (E)DI ← (E)DI + 1;
    ELSE
      (E)SI ← (E)SI – 1;
      (E)DI ← (E)DI – 1;
    FI;
  ELSE IF (word comparison)
    THEN IF DF ← 0
      (E)SI ← (E)SI + 2;
      (E)DI ← (E)DI + 2;
    ELSE
      (E)SI ← (E)SI – 2;
      (E)DI ← (E)DI – 2;
    FI;
  ELSE (* doubleword comparison*)
    THEN IF DF ← 0
      (E)SI ← (E)SI + 4;
      (E)DI ← (E)DI + 4;
    ELSE
      (E)SI ← (E)SI – 4;
      (E)DI ← (E)DI – 4;
    FI;
  FI;
```

## CMPS/CMPSB/CMPSW/CMPD—Compare String Operands (Continued)

### 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.

### 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.

## CMPSD—Compare Scalar Double-Precision Floating-Point Values

Opcode	Instruction	Description
F2 0F C2 /r ib	CMPSD <i>xmm1</i> , <i>xmm2/m64</i> , <i>imm8</i>	Compare low double-precision floating-point value in <i>xmm2/m64</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.

### 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 quadword mask of all 1s (comparison true) or all 0s (comparison false). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand; the high quadword remains unchanged. The comparison predicate operand is an 8-bit immediate the first 3 bits of which define the type of comparison to be made (see Table 3-5); bits 4 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN or in an undefined format. The ordered relationship is true when neither source operand is a NaN or in an undefined format.

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.

Some of the comparisons listed in Table 3-5 can be achieved only through software emulation. For these comparisons 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 3-5 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction.

Pseudo-Op	Implementation
CMPEQSD <i>xmm1</i> , <i>xmm2</i>	CMPSD <i>xmm1</i> , <i>xmm2</i> , 0
CMPLTSD <i>xmm1</i> , <i>xmm2</i>	CMPSD <i>xmm1</i> , <i>xmm2</i> , 1
CMPLESD <i>xmm1</i> , <i>xmm2</i>	CMPSD <i>xmm1</i> , <i>xmm2</i> , 2
CMPUNORDSD <i>xmm1</i> , <i>xmm2</i>	CMPSD <i>xmm1</i> , <i>xmm2</i> , 3
CMPNEQSD <i>xmm1</i> , <i>xmm2</i>	CMPSD <i>xmm1</i> , <i>xmm2</i> , 4
CMPNLTSD <i>xmm1</i> , <i>xmm2</i>	CMPSD <i>xmm1</i> , <i>xmm2</i> , 5
CMPNLESD <i>xmm1</i> , <i>xmm2</i>	CMPSD <i>xmm1</i> , <i>xmm2</i> , 6
CMPORDSD <i>xmm1</i> , <i>xmm2</i>	CMPSD <i>xmm1</i> , <i>xmm2</i> , 7



## CMPD—Compare Scalar Double-Precision Floating-Point Values (Continued)

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.)

### Operation

CASE (COMPARISON PREDICATE) OF

0: OP ← EQ;

1: OP ← LT;

2: OP ← LE;

3: OP ← UNORD;

4: OP ← NEQ;

5: OP ← NLT;

6: OP ← NLE;

7: OP ← ORD;

DEFAULT: Reserved;

CMP0 ← DEST[63-0] OP SRC[63-0];

IF CMP0 == TRUE

THEN DEST[63-0] ← FFFFFFFFFFFFFFFFH

ELSE DEST[63-0] ← 000000000000000H; FI;

\* DEST[127-64] remains unchanged \*;

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

CMPD for equality `__m128d __mm_cmpeq_sd(__m128d a, __m128d b)`

CMPD for less-than `__m128d __mm_cmplt_sd(__m128d a, __m128d b)`

CMPD for less-than-or-equal `__m128d __mm_cmple_sd(__m128d a, __m128d b)`

CMPD for greater-than `__m128d __mm_cmpgt_sd(__m128d a, __m128d b)`

CMPD for greater-than-or-equal `__m128d __mm_cmpge_sd(__m128d a, __m128d b)`

CMPD for inequality `__m128d __mm_cmpneq_sd(__m128d a, __m128d b)`

CMPD for not-less-than `__m128d __mm_cmpnlt_sd(__m128d a, __m128d b)`

CMPD for not-greater-than `__m128d __mm_cmpngt_sd(__m128d a, __m128d b)`

CMPD for not-greater-than-or-equal `__m128d __mm_cmpnge_sd(__m128d a, __m128d b)`

CMPD for ordered `__m128d __mm_cmpord_sd(__m128d a, __m128d b)`

CMPD for unordered `__m128d __mm_cmpunord_sd(__m128d a, __m128d b)`

CMPD for not-less-than-or-equal `__m128d __mm_cmpnle_sd(__m128d a, __m128d b)`

## CMPSD—Compare Scalar Double-Precision Floating-Point Values (Continued)

### SIMD Floating-Point Exceptions

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

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

## **CMPD—Compare Scalar Double-Precision Floating-Point Values (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

#AC(0)                If alignment checking is enabled and an unaligned memory reference is made.

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

Opcode	Instruction	Description
F3 0F C2 /r ib	CMPSS <i>xmm1</i> , <i>xmm2/m32</i> , <i>imm8</i>	Compare low single-precision floating-point value in <i>xmm2/m32</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.

### 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 doubleword mask of all 1s (comparison true) or all 0s (comparison false). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand; the 3 high-order doublewords remain unchanged. The comparison predicate operand is an 8-bit immediate the first 3 bits of which define the type of comparison to be made (see Table 3-5); bits 4 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN or in an undefined format. The ordered relationship is true when neither source operand is a NaN or in an undefined format.

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.

Some of the comparisons listed in Table 3-5 can be achieved only through software emulation. For these comparisons 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 3-5 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction.

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

## CMPSS—Compare Scalar Single-Precision Floating-Point Values (Continued)

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.)

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP ← EQ;
- 1: OP ← LT;
- 2: OP ← LE;
- 3: OP ← UNORD;
- 4: OP ← NEQ;
- 5: OP ← NLT;
- 6: OP ← NLE;
- 7: OP ← ORD;

DEFAULT: Reserved;

CMP0 ← DEST[31-0] OP SRC[31-0];

IF CMP0 == TRUE

THEN DEST[31-0] ← FFFFFFFFH

ELSE DEST[31-0] ← 00000000H; FI;

\* DEST[127-32] remains unchanged \*;

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

CMPSS for equality	<code>__m128 _mm_cmpeq_ss(__m128 a, __m128 b)</code>
CMPSS for less-than	<code>__m128 _mm_cmplt_ss(__m128 a, __m128 b)</code>
CMPSS for less-than-or-equal	<code>__m128 _mm_cmple_ss(__m128 a, __m128 b)</code>
CMPSS for greater-than	<code>__m128 _mm_cmpgt_ss(__m128 a, __m128 b)</code>
CMPSS for greater-than-or-equal	<code>__m128 _mm_cmpge_ss(__m128 a, __m128 b)</code>
CMPSS for inequality	<code>__m128 _mm_cmpneq_ss(__m128 a, __m128 b)</code>
CMPSS for not-less-than	<code>__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)</code>
CMPSS for not-greater-than	<code>__m128 _mm_cmpngt_ss(__m128 a, __m128 b)</code>
CMPSS for not-greater-than-or-equal	<code>__m128 _mm_cmpnge_ss(__m128 a, __m128 b)</code>
CMPSS for ordered	<code>__m128 _mm_cmpord_ss(__m128 a, __m128 b)</code>
CMPSS for unordered	<code>__m128 _mm_cmpunord_ss(__m128 a, __m128 b)</code>
CMPSS for not-less-than-or-equal	<code>__m128 _mm_cmpnle_ss(__m128 a, __m128 b)</code>

## CMPSS—Compare Scalar Single-Precision Floating-Point Values (Continued)

### SIMD Floating-Point Exceptions

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

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.
#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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## **CMPSS—Compare Scalar Single-Precision Floating-Point Values (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

- |                 |   |
|-----------------|---|
| #PF(fault-code) | For a page fault.   |
| #AC(0)          | If alignment checking is enabled and an unaligned memory reference is made. |

## CMPXCHG—Compare and Exchange

Opcode	Instruction	Description
0F B0/ <i>r</i>	CMPXCHG <i>r/m8,r8</i>	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.
0F B1/ <i>r</i>	CMPXCHG <i>r/m16,r16</i>	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 AL.
0F B1/ <i>r</i>	CMPXCHG <i>r/m32,r32</i>	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 AL.

### Description

Compares the value in the AL, AX, or EAX register (depending on the size of the operand) 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, or EAX register.

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.)

### IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

### Operation

(\* accumulator ← AL, AX, or EAX, depending on whether \*)

(\* a byte, word, or doubleword comparison is being performed\*)

IF accumulator ← DEST

THEN

ZF ← 1

DEST ← SRC

ELSE

ZF ← 0

accumulator ← DEST

FI;

### Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.



## CMPXCHG—Compare and Exchange (Continued)

### Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable 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

#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.

## CMPXCHG8B—Compare and Exchange 8 Bytes

Opcode	Instruction	Description
0F C7 /1 m64	CMPXCHG8B <i>m64</i>	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.

### Description

Compares the 64-bit value in EDX:EAX with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX. The destination operand is an 8-byte memory location. 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.

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.)

### IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Pentium processors.

### Operation

```
IF (EDX:EAX ← DEST)
    ZF ← 1
    DEST ← ECX:EBX
ELSE
    ZF ← 0
    EDX:EAX ← DEST
```

### 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.

## CMPXCHG8B—Compare and Exchange 8 Bytes (Continued)

### Protected Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If the destination is located in a nonwritable 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.

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

Opcode	Instruction	Description
66 0F 2F /r	COMISD <i>xmm1</i> , <i>xmm2/m64</i>	Compare low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.

### Description

Compares 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).

Source operand 1 is an XMM register; source 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.

### 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)

## COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS (Continued)

### SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

## COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS (Continued)

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

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

Opcode	Instruction	Description
0F 2F /r	COMISS <i>xmm1</i> , <i>xmm2/m32</i>	Compare low single-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.

### Description

Compares the single-precision floating-point values in the low doublewords of 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).

Source operand 1 is an XMM register; source 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.

### 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)
```

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

### SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.



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

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CPUID—CPU Identification

Opcode	Instruction	Description
0F A2	CPUID	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, according to the input value entered initially in the EAX register

### Description

Returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers. The information returned is selected by entering a value in the EAX register before the instruction is executed. Table 3-6 shows the information returned, depending on the initial value loaded into the EAX register.

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.

The information returned with the CPUID instruction is divided into two groups: basic information and extended function information. Basic information is returned by entering an input value of from 0 to 3 in the EAX register depending on the IA-32 processor type; extended function information is returned by entering an input value of from 80000000H to 80000004H. The extended function CPUID information was introduced with the Pentium 4 processors and is not available in earlier IA-32 processors. Table 3-7 shows the maximum input value that the processor recognizes for the CPUID instruction for basic information and for extended function information, for each family of IA-32 processors on which the CPUID instruction is implemented.

If a higher value than is shown in Table 3-6 is entered for a particular processor, the information for the highest useful basic information value is returned. For example, if an input value of 5 is entered in EAX for a Pentium 4 processor, the information for an input value of 2 is returned. The exception to this rule is the input values that return extended function information (currently, the values 80000000H through 80000004H). For a Pentium 4 processor, entering an input value of 80000005H or above, returns the information for an input value of 2.

The CPUID instruction 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 “Serializing Instructions” in Chapter 7 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*).

When the input value in the EAX register is 0, the processor returns the highest value the CPUID instruction recognizes in the EAX register for returning basic CPUID information (see Table 3-7). A vendor identification string is returned in the EBX, EDX, and ECX registers. For Intel processors, the vendor identification string is “GenuineIntel” as follows:

```
EBX ← 756e6547h (* "Genu", with G in the low nibble of BL *)
EDX ← 49656e69h (* "ineI", with i in the low nibble of DL *)
ECX ← 6c65746eh (* "ntel", with n in the low nibble of CL *)
```

**CPUID—CPU Identification (Continued)**
**Table 3-6. Information Returned by CPUID Instruction**

Initial EAX Value	Information Provided about the Processor	
Basic CPUID Information		
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information (see Table 3-7). “Genu” “ntel” “inel”
1H	EAX EBX  ECX EDX	Version Information (Type, Family, Model, and Stepping ID) Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size. (Value returned * 8 = cache line size) Bits 23-16: Reserved. Bits 31-24: Processor local APIC physical ID Reserved Feature Information (see Figure 3-4 and Table 3-9)
2H	EAX EBX ECX EDX	Cache and TLB Information Cache and TLB Information Cache and TLB Information Cache and TLB Information
3H	EAX EBX ECX EDX	Reserved. Reserved. Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
Extended Function CPUID Information		
8000000H	EAX EBX ECX EDX	Maximum Input Value for Extended Function CPUID Information (see Table 3-7). Reserved. Reserved. Reserved.
8000001H	EAX EBX ECX EDX	Extended Processor Signature and Extended Feature Bits. (Currently Reserved.) Reserved. Reserved. Reserved.
8000002H	EAX EBX ECX EDX	Processor Brand String. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
8000003H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.

## CPUID—CPU Identification (Continued)

**Table 3-6. Information Returned by CPUID Instruction (Continued)**

Initial EAX Value	Information Provided about the Processor	
80000004H	EAX	Processor Brand String Continued.
	EBX	Processor Brand String Continued.
	ECX	Processor Brand String Continued.
	EDX	Processor Brand String Continued.

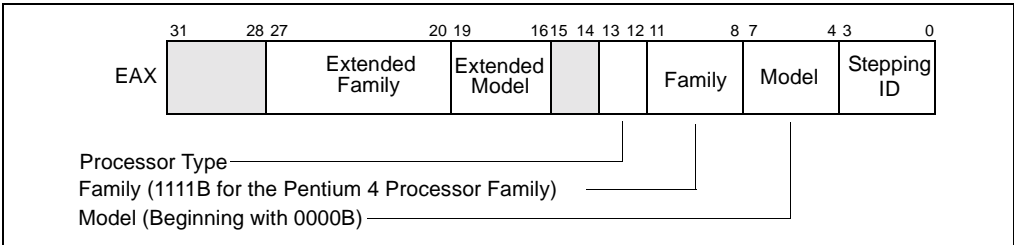
**Table 3-7. Highest CPUID Source Operand for IA-32 Processors**

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	1H	Not Implemented
Pentium Pro and Pentium II Processors, Intel Celeron™ Processors	2H	Not Implemented
Pentium III Processors	3H	Not Implemented
Pentium 4 Processors	2H	80000004H

When the input value is 1, the processor returns version information in the EAX register (see Figure 3-3). The version information consists of an IA-32 processor family identifier, a model identifier, a stepping ID, and a processor type. The model, family, and processor type for the first processor in the Intel Pentium 4 family is as follows:

- Model—0000B
- Family—1111B
- Processor Type—00B

The available processor types are given in Table 3-8. Intel releases information on stepping IDs as needed.



**Figure 3-3. Version Information in the EAX Register**

## CPUID—CPU Identification (Continued)

**Table 3-8. Processor Type Field**

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B
Dual processor*	10B
Intel reserved.	11B

**NOTE:**

\* Not applicable to Intel486 processors.

If the values in the family and/or model fields reach or exceed FH, the CPUID instruction will generate two additional fields in the EAX register: the extended family field and the extended model field. Here, a value of FH in either the model field or the family field indicates that the extended model or family field, respectively, is valid. Family and model numbers beyond FH range from 0FH to FFH, with the least significant hexadecimal digit always FH.

See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) and Chapter 13 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information on identifying earlier IA-32 processors.

When the input value in EAX is 1, three unrelated pieces of information are 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. See “Brand Identification” later in the description of this instruction for information about the intended use of brand indices. This field was introduced in the Pentium® III Xeon™ processors.
- 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 with the Pentium 4 processors.
- Initial APIC ID (high byte of EBX)—this number is the 8-bit physical ID that is assigned to the local APIC on the processor during power up. This field was introduced with the Pentium 4 processors.

### CPUID—CPU Identification (Continued)

When the input value in EAX is 1, feature information is returned to the EDX register (see Figure 3-4). The feature bits permit operating system or application code to determine which IA-32 architectural features are available in the processor. Table 3-9 shows the encoding of the feature flags in the EDX register. For all the feature flags currently returned in EDX, a 1 indicates that the corresponding feature is supported. Software should identify Intel as the vendor to properly interpret the feature flags. (Software should not depend on a 1 indicating the presence of a feature for future feature flags.)

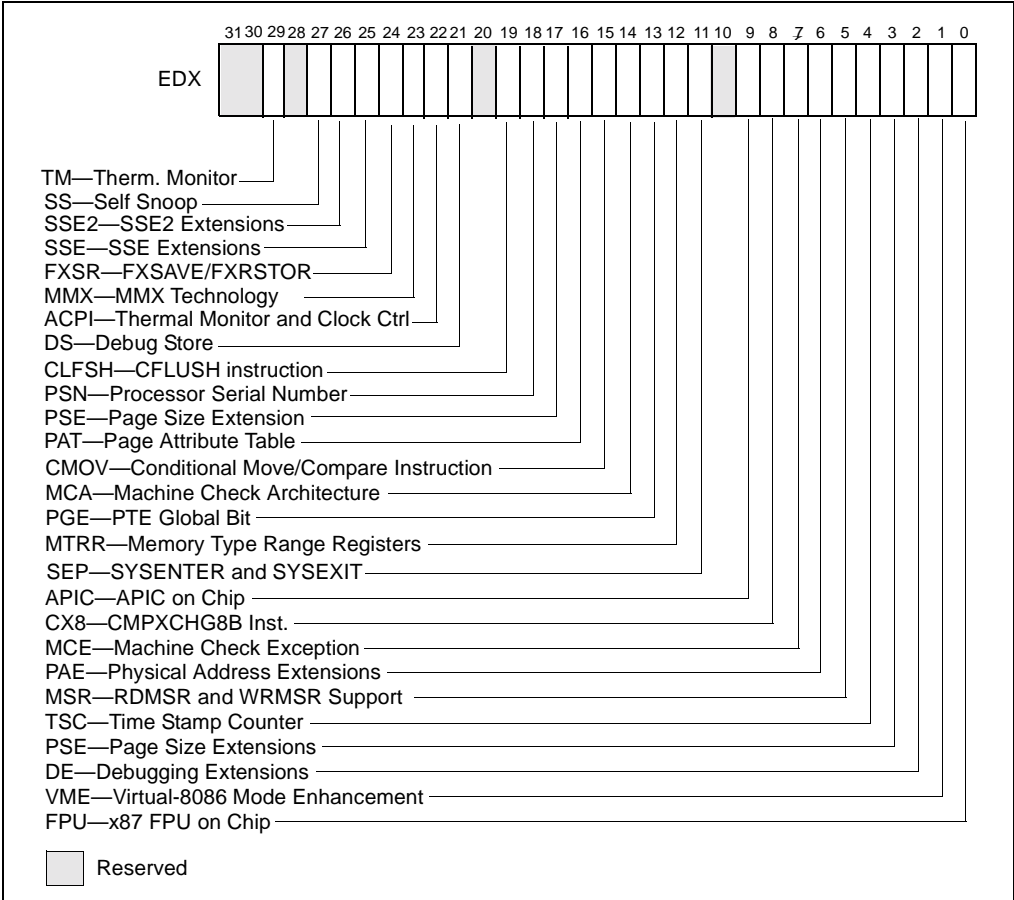


Figure 3-4. Feature Information in the EDX Register

## CPUID—CPU Identification (Continued)

**Table 3-9. CPUID Feature Flags Returned in 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 4Mbyte 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. The actual number of address bits beyond 32 is not defined, and is implementation specific.
7	MCE	<b>Machine Check Exception.</b> Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	<b>CMPXCHG8B Instruction.</b> The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	<b>APIC On-Chip.</b> The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	<b>SYSENTER and SYSEXIT Instructions.</b> The SYSENTER and SYSEXIT and associated 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>PTE Global Bit.</b> The global bit in page directory entries (PDEs) and page table entries (PTEs) is supported, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.

## CPUID—CPU Identification (Continued)

**Table 3-9. CPUID Feature Flags Returned in EDX Register (Continued)**

Bit #	Mnemonic	Description
14	MCA	<b>Machine Check Architecture.</b> The Machine Check Architecture, which provides a compatible mechanism for error reporting in Pentium 4 processors, P6 family processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	<b>Conditional Move Instructions.</b> The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	<b>Page Attribute Table.</b> Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address.
17	PSE-36	<b>32-Bit Page Size Extension.</b> Extended 4-MByte pages that are capable of addressing physical memory beyond 4 GBytes are supported. This feature indicates that the upper four bits of the physical address of the 4-MByte page is encoded by bits 13-16 of the page directory entry.
18	PSN	<b>Processor Serial Number.</b> The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	<b>CLFLUSH Instruction.</b> CLFLUSH Instruction is supported.
20	Reserved	Reserved
21	DS	<b>Debug Store.</b> The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 14, <i>Debugging and Performance Monitoring</i> , in the <i>IA-32 Intel Architecture Software Developer's Manual, Volume 3</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	Reserved	Reserved
29	TM	<b>Thermal Monitor.</b> The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30 - 31	Reserved	Reserved



## CPUID—CPU Identification (Continued)

When the input value is 2, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers. The encoding of these registers is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs. The first member of the family of Pentium 4 processors will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (cleared to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 3-10 shows the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache or TLB types. The descriptors may appear in any order.

**Table 3-10. Encoding of Cache and TLB Descriptors**

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4K-Byte Pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4M-Byte Pages, 4-way set associative, 2 entries
03H	Data TLB: 4K-Byte Pages, 4-way set associative, 64 entries
04H	Data TLB: 4M-Byte Pages, 4-way set associative, 8 entries
06H	1st-level instruction cache: 8K Bytes, 4-way set associative, 32 byte line size
08H	1st-level instruction cache: 16K Bytes, 4-way set associative, 32 byte line size
0AH	1st-level data cache: 8K Bytes, 2-way set associative, 32 byte line size
0CH	1st-level data cache: 16K Bytes, 4-way set associative, 32 byte line size
40H	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	2nd-level cache: 128K Bytes, 4-way set associative, 32 byte line size
42H	2nd-level cache: 256K Bytes, 4-way set associative, 32 byte line size
43H	2nd-level cache: 512K Bytes, 4-way set associative, 32 byte line size
44H	2nd-level cache: 1M Byte, 4-way set associative, 32 byte line size
45H	2nd-level cache: 2M Byte, 4-way set associative, 32 byte line size
50H	Instruction TLB: 4-KByte and 2-MByte or 4-MByte pages, 64 entries
51H	Instruction TLB: 4-KByte and 2-MByte or 4-MByte pages, 128 entries
52H	Instruction TLB: 4-KByte and 2-MByte or 4-MByte pages, 256 entries

## CPUID—CPU Identification (Continued)

**Table 3-10. Encoding of Cache and TLB Descriptors (Continued)**

Descriptor Value	Cache or TLB Description
5BH	Data TLB: 4-KByte and 4-MByte pages, 64 entries
5CH	Data TLB: 4-KByte and 4-MByte pages, 128 entries
5DH	Data TLB: 4-KByte and 4-MByte pages, 256 entries
66H	1st-level data cache: 8KB, 4-way set associative, 64 byte line size
67H	1st-level data cache: 16KB, 4-way set associative, 64 byte line size
68H	1st-level data cache: 32KB, 4-way set associative, 64 byte line size
70H	Trace cache: 12K- $\mu$ op, 8-way set associative
71H	Trace cache: 16K- $\mu$ op, 8-way set associative
72H	Trace cache: 32K- $\mu$ op, 8-way set associative
79H	2nd-level cache: 128KB, 8-way set associative, sectored, 64 byte line size
7AH	2nd-level cache: 256KB, 8-way set associative, sectored, 64 byte line size
7BH	2nd-level cache: 512KB, 8-way set associative, sectored, 64 byte line size
7CH	2nd-level cache: 1MB, 8-way set associative, sectored, 64 byte line size
82H	2nd-level cache: 256K Bytes, 8-way set associative, 32 byte line size
84H	2nd-level cache: 1M Byte, 8-way set associative, 32 byte line size
85H	2nd-level cache: 2M Byte, 8-way set associative, 32 byte line size

The first member of the family of Pentium 4 processors will return the following information about caches and TLBs when the CPUID instruction is executed with an input value of 2:

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

These values are interpreted as follows:

- The least-significant byte (byte 0) of register EAX is set to 01H, indicating that the CPUID instruction needs to be executed only once with an input value of 2 to retrieve complete information about the processor's 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.

## CPUID—CPU Identification (Continued)

- Bytes 1, 2, and 3 of register EAX indicate that the processor contains the following:
  - 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 contains the following:
  - 00H—Null descriptor.
  - 70H—A 12-KByte 1st level code cache, 4-way set associative, with a 64-byte cache line size.
  - 7AH—A 256-KByte 2nd level cache, 8-way set associative, with a 128-byte cache line size.
  - 00H—Null descriptor.

### Brand Identification

To facilitate brand identification of IA-32 processors with the CPUID instruction, two features are provided: brand index and brand string.

The brand index was added to the CPUID instruction with the Pentium III Xeon processor and will be included on all future IA-32 processors, including the Pentium 4 processors. The brand index 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 (for example, “Intel Pentium III processor”).

When executed with a value of 1 in the EAX register, the CPUID instruction returns the 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. Table 3-11 shows those brand indices that currently have processor brand identification strings associated with them.

It is recommended that (1) all reserved entries included in the brand identification table be associated with a brand string that indicates that the index is reserved for future Intel processors and (2) that software be prepared to handle reserved brand indices gracefully.

## CPUID—CPU Identification (Continued)

**Table 3-11. Mapping of Brand Indices and IA-32 Processor Brand Strings**

Brand Index	Brand String
0	This processor does not support the brand identification feature
1	Celeron processor†
2	Pentium III processor†
3	Intel Pentium III Xeon processor
4 – 7	Reserved for future processor
8	Intel Pentium 4 processor
5 – 255	Reserved for future processor

### Note

† Indicates versions of these processors that were introduced after the Pentium III Xeon processor.

The brand string feature is an extension to the CPUID instruction introduced in the Pentium 4 processors. With this feature, the CPUID instruction returns the ASCII brand identification string and the maximum operating frequency of the processor to the EAX, EBX, ECX, and EDX registers. (Note that the frequency returned is the maximum operating frequency that the processor has been qualified for and not the current operating frequency of the processor.)

To use the brand string feature, the CPUID instructions must be executed three times, once with an input value of 8000002H in the EAX register, and a second time an input value of 80000003, and a third time with a value of 80000004H.

The brand string is architecturally defined to be 48 byte long: the first 47 bytes contain ASCII characters and the 48<sup>th</sup> byte is defined to be null (0). The string may be right justified (with leading spaces) for implementation simplicity. For each input value (EAX is 80000002H, 80000003H, or 80000004H), the CPUID instruction returns 16 bytes of the brand string to the EAX, EBX, ECX, and EDX registers. Processor implementations may return less than the 47 ASCII characters, in which case the string will be null terminated and the processor will return valid data for each of the CPUID input values of 80000002H, 80000003H, and 80000004H.

Table 3-12 shows the brand string that is returned by the first processor in the family of Pentium 4 processors.

### NOTE

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

## CPUID—CPU Identification (Continued)

The following procedure can be used for detection of the brand string feature:

1. Execute the CPUID instruction with input value in EAX of 80000000H.
2. If ((EAX\_Return\_Value) AND (80000000H) ≠ 0) then the processor supports the extended CPUID functions and EAX contains the largest extended function input value supported.
3. If EAX\_Return\_Value ≥ 80000004H, then the CPUID instruction supports the brand string feature.

**Table 3-12. Processor Brand String Returned with First Pentium 4 Processor**

EAX Input Value	Return Values	ASCII Equivalent
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"

To identify an IA-32 processor using the CPUID instruction, brand identification software should use the following brand identification techniques ordered by decreasing priority:

- Processor brand string
- Processor brand index and a software supplied brand string table.
- Table based mechanism using type, family, model, stepping, and cache information returned by the CPUID instruction.

### IA-32 Architecture Compatibility

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

### Operation

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;

**CPUID—CPU Identification (Continued)**

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[23:20] ← Extended Family;  
EAX[31:24] ← Reserved;  
EBX[7:0] ← Brand Index;  
EBX[15:8] ← CLFLUSH Line Size;  
EBX[16:23] ← Reserved;  
EBX[24:31] ← Initial APIC ID;  
ECX ← Reserved;  
EDX ← Feature flags; (\* See Figure 3-4 \*)

BREAK;

EAX ← 2H:

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

BREAK;

EAX ← 3H:

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

BREAK;

EAX ← 80000000H:

EAX ← highest extended function input value understood by CPUID;  
EBX ← Reserved;  
ECX ← Reserved;  
EDX ← Reserved;

BREAK;

EAX ← 80000001H:

EAX ← Extended Processor Signature and Feature Bits (\*Currently Reserved\*);  
EBX ← Reserved;  
ECX ← Reserved;  
EDX ← Reserved;

BREAK;

EAX ← 80000002H:

EAX ← Processor Name;  
EBX ← Processor Name;  
ECX ← Processor Name;

## CPUID—CPU Identification (Continued)

```
    EDX ← Processor Name;
BREAK;
EAX ← 80000003H:
    EAX ← Processor Name;
    EBX ← Processor Name;
    ECX ← Processor Name;
    EDX ← Processor Name;
BREAK;
EAX ← 80000004H:
    EAX ← Processor Name;
    EBX ← Processor Name;
    ECX ← Processor Name;
    EDX ← Processor Name;
BREAK;
DEFAULT: (* EAX > highest value recognized by CPUID *)
    EAX ← Reserved; (* undefined*)
    EBX ← Reserved; (* undefined*)
    ECX ← Reserved; (* undefined*)
    EDX ← Reserved; (* undefined*)
BREAK;
ESAC;
```

### Flags Affected

None.

### Exceptions (All Operating Modes)

None.

#### NOTE

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.

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

Opcode	Instruction	Description
F3 0F E6	CVTDQ2PD <i>xmm1</i> , <i>xmm2/m64</i>	Convert two packed signed doubleword integers from <i>xmm2/m128</i> to two packed double-precision floating-point values in <i>xmm1</i> .

### 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 XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the packed integers are located in the low quadword of the register.

### 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

```
CVTDQ2PD    __m128d _mm_cvtepi32_pd(__m128di 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.



## CVTDDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values (Continued)

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

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

Opcode	Instruction	Description
0F 5B /r	CVTDQ2PS <i>xmm1</i> , <i>xmm2/m128</i>	Convert four packed signed doubleword integers from <i>xmm2/m128</i> to four packed single-precision floating-point values in <i>xmm1</i> .

### 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). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. When a conversion is inexact, rounding is performed according to the rounding control bits in the MXCSR register.

### 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]);
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]);
```

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

```
CVTDQ2PS    __m128d _mm_cvtepi32_ps(__m128di a)
```

### SIMD Floating-Point Exceptions

Precision.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

## CVTDDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values (Continued)

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode	Instruction	Description
F2 0F E6	CVTPD2DQ <i>xmm1</i> , <i>xmm2/m128</i>	Convert two packed double-precision floating-point values from <i>xmm2/m128</i> to two packed signed doubleword integers in <i>xmm1</i> .

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 indefinite integer value (80000000H) is returned.

### Operation

```
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] ← 0000000000000000H;
```

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

```
CVTPD2DQ    __m128d __mm_cvtpd_epi32(__m128d a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

## CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
-----	--

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode	Instruction	Description
66 0F 2D /r	CVTPD2PI <i>mm, xmm/m128</i>	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> .

### 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 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 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.

### Operation

```
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]);
```

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

```
CVTPD1PI    __m64 __mm_cvtpd_pi32(__m128d a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#MF	If there is a pending x87 FPU exception.

## CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 5A /r	CVTPD2PS <i>xmm1</i> , <i>xmm2/m128</i>	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> .

### 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). 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.

### Operation

```
DEST[31-0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63-0]);
DEST[63-32] ← Convert_Double_Precision_To_Single_Precision_
               Floating_Point(SRC[127-64]);
DEST[127-64] ← 0000000000000000H;
```

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

```
CVTPD2PS    __m128d _mm_cvtpd_ps(__m128d a)
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.



## CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values (Continued)

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
-----	--

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 2A /r	CVTPI2PD <i>xmm, mm/m64</i>	Convert two packed signed doubleword integers from <i>mm/mem64</i> to two packed double-precision floating-point values in <i>xmm</i> .

### 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 register or a 64-bit memory location. 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 CVTPI2PD instruction is executed.

### 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.

### 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.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

## CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values (Continued)

#UD	<p>If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.</p> <p>If EM in CR0 is set.</p> <p>If OSFXSR in CR4 is 0.</p> <p>If CPUID feature flag SSE2 is 0.</p>
#AC(0)	<p>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</p>

### Real-Address Mode Exceptions

Interrupt 13	<p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p>
#NM	<p>If TS in CR0 is set.</p>
#MF	<p>If there is a pending x87 FPU exception.</p>
#XM	<p>If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.</p>
#UD	<p>If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.</p> <p>If EM in CR0 is set.</p> <p>If OSFXSR in CR4 is 0.</p> <p>If CPUID feature flag SSE2 is 0.</p>

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	<p>For a page fault.</p>
#AC(0)	<p>If alignment checking is enabled and an unaligned memory reference is made.</p>

## CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 2A /r	CVTPI2PS <i>xmm, mm/m64</i>	Convert two signed doubleword integers from <i>mm/m64</i> to two single-precision floating-point values in <i>xmm</i> .

### 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 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.

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.

### 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 remains unchanged \*;

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

CVTPI2PS      `__m128 __mm_cvtpi32_ps(__m128 a, __m64 b)`

### SIMD Floating-Point Exceptions

Precision.

### 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.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

## CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values (Continued)

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers

Opcode	Instruction	Description
66 0F 5B /r	CVTPS2DQ <i>xmm1</i> , <i>xmm2/m128</i>	Convert four packed single-precision floating-point values from <i>xmm2/m128</i> to four packed signed doubleword integers in <i>xmm1</i> .

### Description

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

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 indefinite integer value (80000000H) is returned.

### 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]);
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]);
```

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

```
__m128d _mm_cvtps_epi32(__m128d a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#MF	If there is a pending x87 FPU exception.
#NM	If TS in CR0 is set.

## CVTQPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
0F 5A /r	CVTPS2PD <i>xmm1</i> , <i>xmm2/m64</i>	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> .

### Description

Converts two packed single-precision floating-point values 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 XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the packed single-precision floating-point values are contained in the low quadword of the register.

### Operation

```
DEST[63-0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31-0]);
DEST[127-64] ← Convert_Single_Precision_To_Double_Precision_
                Floating_Point(SRC[63-32]);
```

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

```
CVTPD2PS    __m128d __mm_cvtps_pd(__m128 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0.



## CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values (Continued)

If CPUID feature flag SSE2 is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers

Opcode	Instruction	Description
0F 2D /r	CVTPS2PI <i>mm, xmm/m64</i>	Convert two packed single-precision floating-point values from <i>xmm/m64</i> to two packed signed doubleword integers in <i>mm</i> .

### 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 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 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 CVTPS2PI instruction is executed.

### 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

`__m64 __mm_cvtps_pi32(__m128 a)`

### SIMD Floating-Point Exceptions

Invalid, Precision.

### 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.
#MF	If there is a pending x87 FPU exception.

## CVTQPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

Opcode	Instruction	Description
F2 0F 2D /r	CVTSD2SI <i>r32, xmm/m64</i>	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed doubleword integer <i>r32</i> .

### 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 indefinite integer value (80000000H) is returned.

### Operation

DEST[31-0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63-0]);

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

`int_mm_cvtsd_si32(__m128d a)`

### SIMD Floating-Point Exceptions

Invalid, Precision.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0.

## CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer (Continued)

If CPUID feature flag SSE2 is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

Opcode	Instruction	Description
F2 0F 5A /r	CVTSD2SS <i>xmm1</i> , <i>xmm2/m64</i>	Convert one double-precision floating-point value in <i>xmm2/m64</i> to one single-precision floating-point value in <i>xmm1</i> .

### 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.

### Operation

DEST[31-0] ← Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[63-0]);  
 \* DEST[127-32] remains unchanged \*;

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

CVTSD2SS     \_\_m128\_mm\_cvtsd\_ss(\_\_m128d a, \_\_m128d b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set.

## CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

Opcode	Instruction	Description
F2 0F 2A /r	CVTSI2SD <i>xmm</i> , <i>r/m32</i>	Convert one signed doubleword integer from <i>r/m32</i> to one double-precision floating-point value in <i>xmm</i> .

### Description

Converts a signed doubleword integer in the source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a 32-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 left unchanged.

### Operation

DEST[63-0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[31-0]);

\* DEST[127-64] remains unchanged \*;

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

int\_mm\_cvtsd\_si32(\_\_m128d 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.



## CVTSD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value (Continued)

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

Opcode	Instruction	Description
F3 0F 2A /r	CVTSI2SS <i>xmm</i> , <i>r/m32</i>	Convert one signed doubleword integer from <i>r/m32</i> to one single-precision floating-point value in <i>xmm</i> .

### Description

Converts a signed doubleword integer 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 32-bit 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.

### Operation

DEST[31-0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31-0]);  
 \* DEST[127-32] remains unchanged \*;

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

`__m128_mm_cvtsi32_ss(__m128d a, int b)`

### SIMD Floating-Point Exceptions

Precision.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0.

## CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value (Continued)

If CPUID feature flag SSE is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

Opcode	Instruction	Description
F3 0F 5A /r	CVTSS2SD <i>xmm1</i> , <i>xmm2/m32</i>	Convert one single-precision floating-point value in <i>xmm2/m32</i> to one double-precision floating-point value in <i>xmm1</i> .

### 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.

### Operation

DEST[63-0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31-0]);  
 \* DEST[127-64] remains unchanged \*;

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

CVTSS2SD     \_\_m128d\_mm\_cvtsd\_sd(\_\_m128d a, \_\_m128 b)

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.

## CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

Opcode	Instruction	Description
F3 0F 2D /r	CVTSS2SI <i>r32</i> , <i>xmm/m32</i>	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed doubleword integer in <i>r32</i> .

### Description

Converts a single-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 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, 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 indefinite integer value (80000000H) is returned.

### Operation

DEST[31-0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31-0]);

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

int\_mm\_cvtss\_si32(\_\_m128d a)

### SIMD Floating-Point Exceptions

Invalid, Precision.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set.

## CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode	Instruction	Description
66 0F 2C /r	CVTTPD2PI <i>mm, xmm/m128</i>	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.

### 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 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 indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

### Operation

```
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]);
```

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

```
CVTTPD1PI    __m64 __mm_cvttpd_pi32(__m128d a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Protected Mode Exceptions

- |                 |   |
|-----------------|---|
| #GP(0)          | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br><br>If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0)          | For an illegal address in the SS segment.   |
| #PF(fault-code) | For a page fault.   |



## CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

#MF	If there is a pending x87 FPU exception.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode	Instruction	Description
66 0F E6	CVTTPD2DQ <i>xmm1</i> , <i>xmm2/m128</i>	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.

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, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the indefinite integer value (80000000H) is returned.

### Operation

```
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] ← 0000000000000000H;
```

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

```
CVTTPD2DQ  __m128i _mm_cvttpd_epi32(__m128d a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

## CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
-----	--

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers

Opcode	Instruction	Description
F3 0F 5B /r	CVTTPS2DQ <i>xmm1</i> , <i>xmm2/m128</i>	Convert four single-precision floating-point values from <i>xmm2/m128</i> to four signed doubleword integers in <i>xmm1</i> using truncation.

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

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 indefinite integer value (80000000H) is returned.

### 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]);
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]);
```

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

```
__m128d _mm_cvttps_epi32(__m128d a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

## CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
-----	--

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers

Opcode	Instruction	Description
0F 2C /r	CVTTPS2PI <i>mm, xmm/m64</i>	Convert two single-precision floating-point values from <i>xmm/m64</i> to two signed doubleword signed integers in <i>mm</i> using truncation.

### 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 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 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.

### 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

```
__m64 __mm_cvttps_pi32(__m128 a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### 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.
#MF	If there is a pending x87 FPU exception.

## CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer

Opcode	Instruction	Description
F2 0F 2C /r	CVTTSD2SI <i>r32, xmm/m64</i>	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed doubleword integer in <i>r32</i> using truncation.

### 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, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the indefinite integer value (80000000H) is returned.

### Operation

DEST[31-0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63-0]);

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

int\_mm\_cvtsd\_si32(\_\_m128d a)

### SIMD Floating-Point Exceptions

Invalid, Precision.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set.



## CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer

Opcode	Instruction	Description
F3 0F 2C /r	CVTTSS2SI <i>r32, xmm/m32</i>	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed doubleword integer in <i>r32</i> using truncation.

### Description

Converts a single-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 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 indefinite integer value (80000000H) is returned.

### Operation

DEST[31-0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31-0]);

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

int\_mm\_cvtss\_si32(\_\_m128d a)

### SIMD Floating-Point Exceptions

Invalid, Precision.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set.

## CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## CWD/CDQ—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Description
99	CWD	DX:AX ← sign-extend of AX
99	CDQ	EDX:EAX ← sign-extend of EAX

### Description

Doubles the size of the operand in register AX or EAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX or EDX:EAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register (see Figure 7-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword 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.

### Operation

```
IF OperandSize ← 16 (* CWD instruction *)
  THEN DX ← SignExtend(AX);
  ELSE (* OperandSize ← 32, CDQ instruction *)
    EDX ← SignExtend(EAX);
FI;
```

### Flags Affected

None.

### Exceptions (All Operating Modes)

None.

**CWDE—Convert Word to Doubleword**

See entry for CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword.

## DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Description
27	DAA	Decimal adjust AL after addition

### 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.

### Operation

```
IF (((AL AND 0FH) > 9) or AF ← 1)
  THEN
    AL ← AL + 6;
    CF ← CF OR CarryFromLastAddition; (* CF OR carry from AL ← AL + 6 *)
    AF ← 1;
  ELSE
    AF ← 0;
FI;
IF ((AL AND F0H) > 90H) or CF ← 1)
  THEN
    AL ← AL + 60H;
    CF ← 1;
  ELSE
    CF ← 0;
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=04H  BL=35H  EFLAGS(OSZAPC)=X00101
```

## **DAA—Decimal Adjust AL after Addition (Continued)**

### **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.

### **Exceptions (All Operating Modes)**

None.

## DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Description
2F	DAS	Decimal adjust AL after subtraction

### 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.

### Operation

```
IF (AL AND 0FH) > 9 OR AF ← 1
  THEN
    AL ← AL - 6;
    CF ← CF OR BorrowFromLastSubtraction; (* CF OR borrow from AL ← AL - 6 *)
    AF ← 1;
  ELSE AF ← 0;
FI;
IF ((AL > 9FH) or CF ← 1)
  THEN
    AL ← AL - 60H;
    CF ← 1;
  ELSE CF ← 0;
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.

### Exceptions (All Operating Modes)

None.



## DEC—Decrement by 1

Opcode	Instruction	Description
FE /1	DEC <i>r/m8</i>	Decrement <i>r/m8</i> by 1
FF /1	DEC <i>r/m16</i>	Decrement <i>r/m16</i> by 1
FF /1	DEC <i>r/m32</i>	Decrement <i>r/m32</i> by 1
48+rw	DEC <i>r16</i>	Decrement <i>r16</i> by 1
48+rd	DEC <i>r32</i>	Decrement <i>r32</i> by 1

### 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.

### 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 nonwritable 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

#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.

## DEC—Decrement by 1 (Continued)

### 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.

## DIV—Unsigned Divide

Opcode	Instruction	Description
F6 /6	DIV <i>r/m8</i>	Unsigned divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder
F7 /6	DIV <i>r/m16</i>	Unsigned divide DX:AX by <i>r/m16</i> , with result stored in AX ← Quotient, DX ← Remainder
F7 /6	DIV <i>r/m32</i>	Unsigned divide EDX:EAX by <i>r/m32</i> , with result stored in EAX ← Quotient, EDX ← Remainder

### Description

Divides (unsigned) the value in the AX, DX:AX, or EDX:EAX registers (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), as shown in the following table:

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	<i>r/m8</i>	AL	AH	255
Doubleword/word	DX:AX	<i>r/m16</i>	AX	DX	65,535
Quadword/doubleword	EDX:EAX	<i>r/m32</i>	EAX	EDX	$2^{32} - 1$

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.

### Operation

```

IF SRC ← 0
  THEN #DE; (* divide error *)
FI;
IF OperandSize ← 8 (* word/byte operation *)
  THEN
    temp ← AX / SRC;
    IF temp > FFH
      THEN #DE; (* divide error *) ;
    ELSE
      AL ← temp;
      AH ← AX MOD SRC;
    FI;
  
```

**DIV—Unsigned Divide (Continued)**

```

ELSE
  IF OperandSize ← 16 (* doubleword/word operation *)
    THEN
      temp ← DX:AX / SRC;

      IF temp > FFFFH
        THEN #DE; (* divide error *);
        ELSE
          AX ← temp;
          DX ← DX:AX MOD SRC;

          FI;
      ELSE (* quadword/doubleword operation *)
        temp ← EDX:EAX / SRC;
        IF temp > FFFFFFFFH
          THEN #DE; (* divide error *);
          ELSE
            EAX ← temp;
            EDX ← EDX:EAX MOD SRC;

            FI;
      FI;
FI;

```

**Flags Affected**

The CF, OF, SF, ZF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#DE	If the source operand (divisor) is 0. If the quotient is too large for the designated register.
-----	--

**DIV—Unsigned Divide (Continued)**

#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.

**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.

## DIVPD—Divide Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 5E /r	DIVPD <i>xmm1</i> , <i>xmm2/m128</i>	Divide packed double-precision floating-point values in <i>xmm1</i> by packed double-precision floating-point values <i>xmm2/m128</i> .

### Description

Performs a SIMD divide of the two packed double-precision floating-point values in the destination operand (first operand) by the two packed double-precision floating-point values in the source operand (second 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD double-precision floating-point operation.

### Operation

DEST[63-0] ← DEST[63-0] / (SRC[63-0]);  
 DEST[127-64] ← DEST[127-64] / (SRC[127-64]);

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

DIVPD            \_\_m128 \_mm\_div\_pd(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.

## DIVPD—Divide Packed Double-Precision Floating-Point Values (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 5E /r	DIVPS <i>xmm1</i> , <i>xmm2/m128</i>	Divide packed single-precision floating-point values in <i>xmm1</i> by packed single-precision floating-point values <i>xmm2/m128</i> .

### Description

Performs a SIMD divide of the two packed single-precision floating-point values in the destination operand (first operand) by the two 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD single-precision floating-point operation.

### Operation

```
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]);
```

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

```
DIVPS      __m128 _mm_div_ps(__m128 a, __m128 b)
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.



## DIVPS—Divide Packed Single-Precision Floating-Point Values (Continued)

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
-----	---

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## DIVSD—Divide Scalar Double-Precision Floating-Point Values

Opcode	Instruction	Description
F2 0F 5E /r	DIVSD <i>xmm1</i> , <i>xmm2/m64</i>	Divide low double-precision floating-point value in <i>xmm1</i> by low double-precision floating-point value in <i>xmm2/mem64</i> .

### Description

Divides the low double-precision floating-point value in the destination operand (first operand) by 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

### Operation

DEST[63-0] ← DEST[63-0] / SRC[63-0];

\* DEST[127-64] remains unchanged \*;

### 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.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

## DIVSD—Divide Scalar Double-Precision Floating-Point Values (Continued)

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 5E /r	DIVSS <i>xmm1</i> , <i>xmm2/m32</i>	Divide low single-precision floating-point value in <i>xmm1</i> by low single-precision floating-point value in <i>xmm2/m32</i>

### Description

Divides the low single-precision floating-point value in the destination operand (first operand) by 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

### Operation

DEST[31-0] ← DEST[31-0] / SRC[31-0];

\* DEST[127-32] remains unchanged \*;

### 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.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.

## DIVSS—Divide Scalar Single-Precision Floating-Point Values (Continued)

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## EMMS—Empty MMX State

Opcode	Instruction	Description
0F 77	EMMS	Set the x87 FPU tag word to empty.

### 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 registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the *IA-32 Intel Architecture 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 state at the end of all MMX 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.

### Operation

`x87FPUTagWord ← FFFFH;`

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

`void_mm_empty()`

### Flags Affected

None.

### Protected Mode Exceptions

#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

### Real-Address Mode Exceptions

Same as for protected mode exceptions.

### Virtual-8086 Mode Exceptions

Same as for protected mode exceptions.

## ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Description
C8 <i>iw</i> 00	ENTER <i>imm16</i> ,0	Create a stack frame for a procedure
C8 <i>iw</i> 01	ENTER <i>imm16</i> ,1	Create a nested stack frame for a procedure
C8 <i>iw</i> <i>ib</i>	ENTER <i>imm16</i> , <i>imm8</i>	Create a nested stack frame for a procedure

### 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) or EBP (32 bits) register specifies the current frame pointer and whether SP (16 bits) or ESP (32 bits) specifies the stack pointer.

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 EBP register onto the stack, copies the current stack pointer from the ESP register into the EBP register, and loads the ESP 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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for more information about the actions of the ENTER instruction.

### Operation

NestingLevel ← NestingLevel MOD 32

IF StackSize ← 32

THEN

Push(EBP) ;

FrameTemp ← ESP;

ELSE (\* StackSize ← 16\*)

Push(BP);

FrameTemp ← SP;

FI;

IF NestingLevel ← 0

THEN GOTO CONTINUE;

FI;

**ENTER—Make Stack Frame for Procedure Parameters (Continued)**

```

IF (NestingLevel > 0)
  FOR i ← 1 TO (NestingLevel - 1)
    DO
      IF OperandSize ← 32
        THEN
          IF StackSize ← 32
            EBP ← EBP - 4;
            Push([EBP]); (* doubleword push *)
          ELSE (* StackSize ← 16*)
            BP ← BP - 4;
            Push([BP]); (* doubleword push *)
          FI;
        ELSE (* OperandSize ← 16 *)
          IF StackSize ← 32
            THEN
              EBP ← EBP - 2;
              Push([EBP]); (* word push *)
            ELSE (* StackSize ← 16*)
              BP ← BP - 2;
              Push([BP]); (* word push *)
            FI;
          FI;
        OD;
      IF OperandSize ← 32
        THEN
          Push(FrameTemp); (* doubleword push *)
        ELSE (* OperandSize ← 16 *)
          Push(FrameTemp); (* word push *)
        FI;
      GOTO CONTINUE;
    FI;
  CONTINUE:
  IF StackSize ← 32
    THEN
      EBP ← FrameTemp
      ESP ← EBP - Size;
    ELSE (* StackSize ← 16*)
      BP ← FrameTemp
      SP ← BP - Size;
    FI;
  END;

```

**Flags Affected**

None.



## ENTER—Make Stack Frame for Procedure Parameters (Continued)

### 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.

### Real-Address Mode Exceptions

- #SS(0)                    If the new value of the SP or ESP register is outside the stack segment limit.

### 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.

**F2XM1—Compute  $2^x-1$** 

Opcode	Instruction	Description
D9 F0	F2XM1	Replace ST(0) with $(2^{\text{ST}(0)} - 1)$

**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.

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:

$$x^y \leftarrow 2^{(y * \log_2 x)}$$

**Operation**

$$\text{ST}(0) \leftarrow (2^{\text{ST}(0)} - 1);$$

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C0, C2, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Result is a denormal value.

**F2XM1—Compute  $2^x-1$  (Continued)**

- #U                      Result is too small for destination format.
- #P                      Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

- #NM                    EM or TS in CR0 is set.

**Real-Address Mode Exceptions**

- #NM                    EM or TS in CR0 is set.

**Virtual-8086 Mode Exceptions**

- #NM                    EM or TS in CR0 is set.

## FABS—Absolute Value

Opcode	Instruction	Description
D9 E1	FABS	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.

ST(0) SRC	ST(0) DEST
$-\infty$	$+\infty$
-F	+F
-0	+0
+0	+0
+F	+F
$+\infty$	$+\infty$
NaN	NaN

#### NOTE:

F Means finite floating-point value.

### Operation

$$ST(0) \leftarrow |ST(0)|$$

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred; otherwise, cleared to 0.  
 C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

### Protected Mode Exceptions

#NM EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

## FABS—Absolute Value (Continued)

### Virtual-8086 Mode Exceptions

#NM                      EM or TS in CR0 is set.

## FADD/FADDP/FIADD—Add

Opcode	Instruction	Description
D8 /0	FADD <i>m32fp</i>	Add <i>m32fp</i> to ST(0) and store result in ST(0)
DC /0	FADD <i>m64fp</i>	Add <i>m64fp</i> to ST(0) and store result in ST(0)
D8 C0+i	FADD ST(0), ST(i)	Add ST(0) to ST(i) and store result in ST(0)
DC C0+i	FADD ST(i), ST(0)	Add ST(i) to ST(0) and store result in ST(i)
DE C0+i	FADDP ST(i), ST(0)	Add ST(0) to ST(i), store result in ST(i), and pop the register stack
DE C1	FADDP	Add ST(0) to ST(1), store result in ST(1), and pop the register stack
DA /0	FIADD <i>m32int</i>	Add <i>m32int</i> to ST(0) and store result in ST(0)
DE /0	FIADD <i>m16int</i>	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.

## FADD/FADDP/FIADD—Add (Continued)

		DEST						
		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
SRC	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	NaN
	$-F$ or $-I$	$-\infty$	$-F$	SRC	SRC	$\pm F$ or $\pm 0$	$+\infty$	NaN
	$-0$	$-\infty$	DEST	$-0$	$\pm 0$	DEST	$+\infty$	NaN
	$+0$	$-\infty$	DEST	$\pm 0$	$+0$	DEST	$+\infty$	NaN
	$+F$ or $+I$	$-\infty$	$\pm F$ or $\pm 0$	SRC	SRC	$+F$	$+\infty$	NaN
	$+\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

F Means finite floating-point value.

I Means integer.

\* Indicates floating-point invalid-arithmic-operand (#IA) exception.

### Operation

IF instruction is FIADD

THEN

DEST  $\leftarrow$  DEST + ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (\* source operand is floating-point value \*)

DEST  $\leftarrow$  DEST + SRC;

FI;

IF instruction  $\leftarrow$  FADDP

THEN

PopRegisterStack;

FI;

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) is generated: 0  $\leftarrow$  not roundup; 1  $\leftarrow$  roundup.

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.

## FADD/FADDP/FIADD—Add (Continued)

#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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.



## FBLD—Load Binary Coded Decimal

Opcode	Instruction	Description
DF /4	FBLD <i>m80 dec</i>	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.

### Operation

TOP  $\leftarrow$  TOP  $- 1$ ;  
ST(0)  $\leftarrow$  ConvertToDoubleExtendedPrecisionFP(SRC);

### FPU Flags Affected

C1                      Set to 1 if stack overflow occurred; otherwise, cleared 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                    EM or TS in CR0 is set.  
#PF(fault-code)     If a page fault occurs.  
#AC(0)                If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## FBLD—Load Binary Coded Decimal (Continued)

### 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	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FBSTP—Store BCD Integer and Pop

Opcode	Instruction	Description
DF /6	FBSTP m80bcd	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.

ST(0)	DEST
$-\infty$	*
$-F < -1$	-D
$-1 < -F < -0$	**
-0	-0
+0	+0
$+0 < +F < +1$	**
$+F > +1$	+D
$+\infty$	*
NaN	*

#### NOTES:

F Means finite floating-point value.

D Means packed-BCD number.

\* Indicates floating-point invalid-operation (#IA) exception.

\*\*  $\pm 0$  or  $\pm 1$ , depending on the rounding mode.

If the source value is too large for the destination format and the invalid-operation exception is not masked, an invalid-operation exception 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.

If the source value is a quiet NaN, an invalid-operation exception is generated. Quiet NaNs do not normally cause this exception to be generated.

## FBSTP—Store BCD Integer and Pop (Continued)

### Operation

DEST ← BCD(ST(0));  
PopRegisterStack;

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction if the inexact exception (#P) is generated: 0 = not roundup; 1 ← roundup.
C0, C2, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is empty; contains a NaN, $\pm\infty$ , or unsupported format; or contains value that exceeds 18 BCD digits in length.
#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 nonwritable 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

## FBSTP—Store BCD Integer and Pop (Continued)

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FCFS—Change Sign

Opcode	Instruction	Description
D9 E0	FCFS	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.

ST(0) SRC	ST(0) DEST
$-\infty$	$+\infty$
-F	+F
-0	+0
+0	-0
+F	-F
$+\infty$	$-\infty$
NaN	NaN

#### NOTE:

F Means finite floating-point value.

### Operation

$\text{SignBit}(\text{ST}(0)) \leftarrow \text{NOT}(\text{SignBit}(\text{ST}(0)))$

### FPU Flags Affected

C1                      Set to 0 if stack underflow occurred; otherwise, cleared to 0.  
 C0, C2, C3            Undefined.

### Floating-Point Exceptions

#IS                      Stack underflow occurred.

### Protected Mode Exceptions

#NM                      EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM                      EM or TS in CR0 is set.

## FCHS—Change Sign (Continued)

### Virtual-8086 Mode Exceptions

#NM                      EM or TS in CR0 is set.

## FCLEX/FNCLEX—Clear Exceptions

Opcode	Instruction	Description
9B DB E2	FCLEX	Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.
DB E2	FNCLEX*	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

### NOTE:

\* See “IA-32 Architecture Compatibility” 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 in 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 *IA-32 Intel Architecture 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 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.

### 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.



## FCLEX/FNCLEX—Clear Exceptions (Continued)

### Protected Mode Exceptions

#NM                    EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM                    EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM                    EM or TS in CR0 is set.

## FCMOV<sub>cc</sub>—Floating-Point Conditional Move

Opcode	Instruction	Description
DA C0+i	FCMOVB ST(0), ST(i)	Move if below (CF=1)
DA C8+i	FCMOVE ST(0), ST(i)	Move if equal (ZF=1)
DA D0+i	FCMOVBE ST(0), ST(i)	Move if below or equal (CF=1 or ZF=1)
DA D8+i	FCMOVU ST(0), ST(i)	Move if unordered (PF=1)
DB C0+i	FCMOVNB ST(0), ST(i)	Move if not below (CF=0)
DB C8+i	FCMOVNE ST(0), ST(i)	Move if not equal (ZF=0)
DB D0+i	FCMOVNBE ST(0), ST(i)	Move if not below or equal (CF=0 and ZF=0)
DB D8+i	FCMOVNU ST(0), ST(i)	Move if not unordered (PF=0)

### 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 conditions for each mnemonic are given in the Description column above and in Table 7-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOV<sub>cc</sub> instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOV<sub>cc</sub> instructions. Software can check if the FCMOV<sub>cc</sub> instructions are supported by checking the processor's feature information with the CPUID instruction (see "COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS" in this chapter). If both the CMOV and FPU feature bits are set, the FCMOV<sub>cc</sub> instructions are supported.

### IA-32 Architecture Compatibility

The FCMOV<sub>cc</sub> instructions were introduced to the IA-32 Architecture in the Pentium Pro processor family and is not available in earlier IA-32 processors.

### Operation

IF condition TRUE  
 ST(0) ← ST(i)  
 FI;

### FPU Flags Affected

C1                      Set to 0 if stack underflow occurred.  
 C0, C2, C3            Undefined.

## FCMOVcc—Floating-Point Conditional Move (Continued)

### Floating-Point Exceptions

#IS                      Stack underflow occurred.

### Integer Flags Affected

None.

### Protected Mode Exceptions

#NM                      EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM                      EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM                      EM or TS in CR0 is set.

## FCOM/FCOMP/FCOMPP—Compare Floating Point Values

Opcode	Instruction	Description
D8 /2	FCOM <i>m32fp</i>	Compare ST(0) with <i>m32fp</i> .
DC /2	FCOM <i>m64fp</i>	Compare ST(0) with <i>m64fp</i> .
D8 D0+i	FCOM ST(i)	Compare ST(0) with ST(i).
D8 D1	FCOM	Compare ST(0) with ST(1).
D8 /3	FCOMP <i>m32fp</i>	Compare ST(0) with <i>m32fp</i> and pop register stack.
DC /3	FCOMP <i>m64fp</i>	Compare ST(0) with <i>m64fp</i> and pop register stack.
D8 D8+i	FCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	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  $-0.0 \leftarrow +0.0$ .

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) $\leftarrow$ SRC	1	0	0
Unordered*	1	1	1

## NOTE:

\* 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” 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.

## FCOM/FCOMP/FCOMPP—Compare Floating Point Values (Continued)

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.

### Operation

```

CASE (relation of operands) OF
  ST > SRC:      C3, C2, C0 ← 000;
  ST < SRC:      C3, C2, C0 ← 001;
  ST ← SRC:     C3, C2, C0 ← 100;
ESAC;
IF ST(0) or SRC ← NaN or unsupported format
  THEN
    #IA
    IF FPUControlWord.IM ← 1
      THEN
        C3, C2, C0 ← 111;
    FI;
FI;
IF instruction ← FCOMP
  THEN
    PopRegisterStack;
FI;
IF instruction ← FCOMPP
  THEN
    PopRegisterStack;
    PopRegisterStack;
FI;

```

### FPU Flags Affected

C1                    Set to 0 if stack underflow occurred; otherwise, cleared 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.

## FCOM/FCOMP/FCOMPP—Compare Floating Point Values (Continued)

#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 EM or TS in CR0 is set.

#PF(fault-code) If a 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.

#NM EM or TS in CR0 is set.

### 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 EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS

Opcode	Instruction	Description
DB F0+i	FCOMI ST, ST(i)	Compare ST(0) with ST(i) and set status flags accordingly
DF F0+i	FCOMIP ST, ST(i)	Compare ST(0) with ST(i), set status flags accordingly, and pop register stack
DB E8+i	FUCOMI ST, ST(i)	Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly
DF E8+i	FUCOMIP ST, ST(i)	Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack

### Description

Compares the contents of register 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  $-0.0 \leftarrow +0.0$ .

Comparison Results	ZF	PF	CF
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 $\leftarrow$ ST(i)	1	0	0
Unordered*	1	1	1

#### NOTE:

\* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

The FCOMI/FCOMIP instructions perform the same operation as the FUCOMI/FUCOMIP instructions. The only difference is how they handle QNaN operands. The FCOMI/FCOMIP instructions set the status flags to “unordered” and generate an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value (SNaN or QNaN) or is in an unsupported format.

The FUCOMI/FUCOMIP instructions perform the same operation as the FCOMI/FCOMIP instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs. See “FXAM—Examine” in this chapter for additional information on unordered comparisons.

If invalid-operation exception is unmasked, the status flags are not set if the invalid-arithmetic-operand exception is generated.

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.

## FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS (Continued)

### IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the Pentium Pro processor family 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;



## FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS (Continued)

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared 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	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## FCOS—Cosine

Opcode	Instruction	Description
D9 FF	FCOS	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, assuming that neither overflow nor underflow occurs.

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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

### Operation

```
IF |ST(0)| < 263
THEN
    C2 ← 0;
    ST(0) ← cosine(ST(0));
ELSE (*source operand is out-of-range *)
    C2 ← 1;
FI;
```

## FCOS—Cosine (Continued)

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.  Undefined if C2 is 1.
C2	Set to 1 if source operand is outside the range $-2^{63}$ to $+2^{63}$ ; otherwise, cleared to 0.
C0, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Result 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	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

**FDECSTP—Decrement Stack-Top Pointer**

Opcode	Instruction	Description
D9 F6	FDECSTP	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.

**Operation**

```
IF TOP ← 0
  THEN TOP ← 7;
  ELSE TOP ← TOP - 1;
FI;
```

**FPU Flags Affected**

The C1 flag is set to 0; otherwise, cleared to 0. The C0, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM EM or TS in CR0 is set.

**Real-Address Mode Exceptions**

#NM EM or TS in CR0 is set.

**Virtual-8086 Mode Exceptions**

#NM EM or TS in CR0 is set.

## FDIV/FDIVP/FIDIV—Divide

Opcode	Instruction	Description
D8 /6	FDIV <i>m32fp</i>	Divide ST(0) by <i>m32fp</i> and store result in ST(0)
DC /6	FDIV <i>m64fp</i>	Divide ST(0) by <i>m64fp</i> and store result in ST(0)
D8 F0+i	FDIV ST(0), ST(i)	Divide ST(0) by ST(i) and store result in ST(0)
DC F8+i	FDIV ST(i), ST(0)	Divide ST(i) by ST(0) and store result in ST(i)
DE F8+i	FDIVP ST(i), ST(0)	Divide ST(i) by ST(0), store result in ST(i), and pop the register stack
DE F9	FDIVP	Divide ST(1) by ST(0), store result in ST(1), and pop the register stack
DA /6	FIDIV <i>m32int</i>	Divide ST(0) by <i>m32int</i> and store result in ST(0)
DE /6	FIDIV <i>m64int</i>	Divide ST(0) by <i>m64int</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 or in 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.

## FDIV/FDIVP/FIDIV—Divide (Continued)

		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.

## Operation

IF SRC = 0

THEN

#Z

ELSE

IF instruction is FIDIV

THEN

DEST ← DEST / ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (\* source operand is floating-point value \*)

DEST ← DEST / SRC;

FI;

FI;

IF instruction ← FDIVP

THEN

PopRegisterStack

FI;

## FDIV/FDIVP/FIDIV—Divide (Continued)

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C0, C2, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.  $\pm\infty / \pm\infty$ ; $\pm 0 / \pm 0$
#D	Result 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

## FDIV/FDIVP/FIDIV—Divide (Continued)

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.



## FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	Description
D8 /7	FDIVR <i>m32fp</i>	Divide <i>m32fp</i> by ST(0) and store result in ST(0)
DC /7	FDIVR <i>m64fp</i>	Divide <i>m64fp</i> by ST(0) and store result in ST(0)
D8 F8+i	FDIVR ST(0), ST(i)	Divide ST(i) by ST(0) and store result in ST(0)
DC F0+i	FDIVR ST(i), ST(0)	Divide ST(0) by ST(i) and store result in ST(i)
DE F0+i	FDIVRP ST(i), ST(0)	Divide ST(0) by ST(i), store result in ST(i), and pop the register stack
DE F1	FDIVRP	Divide ST(0) by ST(1), store result in ST(1), and pop the register stack
DA /7	FIDIVR <i>m32int</i>	Divide <i>m32int</i> by ST(0) and store result in ST(0)
DE /7	FIDIVR <i>m16int</i>	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 or in 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.

## FDIVR/FDIVRP/FIDIVR—Reverse Divide (Continued)

		DEST						
		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
SRC	$-\infty$	*	$+\infty$	$+\infty$	$-\infty$	$-\infty$	*	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
	$+\infty$	*	$-\infty$	$-\infty$	$+\infty$	$+\infty$	*	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.

**Operation**

IF DEST = 0

THEN

#Z

ELSE

IF instruction is FIDIVR

THEN

DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) / DEST;

ELSE (\* source operand is floating-point value \*)

DEST ← SRC / DEST;

FI;

FI;

IF instruction ← FDIVRP

THEN

PopRegisterStack

FI;

## FDIVR/FDIVRP/FIDIVR—Reverse Divide (Continued)

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C0, C2, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.  $\pm\infty / \pm\infty$ ; $\pm 0 / \pm 0$
#D	Result 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

**FDIVR/FDIVRP/FIDIVR—Reverse Divide (Continued)****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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FFREE—Free Floating-Point Register

Opcode	Instruction	Description
DD C0+i	FFREE ST(i)	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.

### Operation

TAG(i) ← 11B;

### FPU Flags Affected

C0, C1, C2, C3 undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM EM or TS in CR0 is set.

## FICOM/FICOMP—Compare Integer

Opcode	Instruction	Description
DE /2	FICOM <i>m16int</i>	Compare ST(0) with <i>m16int</i>
DA /2	FICOM <i>m32int</i>	Compare ST(0) with <i>m32int</i>
DE /3	FICOMP <i>m16int</i>	Compare ST(0) with <i>m16int</i> and pop stack register
DA /3	FICOMP <i>m32int</i>	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.

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” 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.

### 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;
```

## FICOM/FICOMP—Compare Integer (Continued)

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FILD—Load Integer

Opcode	Instruction	Description
DF /0	FILD <i>m16int</i>	Push <i>m16int</i> onto the FPU register stack.
DB /0	FILD <i>m32int</i>	Push <i>m32int</i> onto the FPU register stack.
DF /5	FILD <i>m64int</i>	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.

### Operation

TOP ← TOP – 1;  
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);

### FPU Flags Affected

C1 Set to 1 if stack overflow occurred; cleared 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 EM or TS in CR0 is set.  
#PF(fault-code) If a page fault occurs.  
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## FILD—Load Integer (Continued)

### 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	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FINCSTP—Increment Stack-Top Pointer

Opcode	Instruction	Description
D9 F7	FINCSTP	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.

### Operation

```
IF TOP ← 7
  THEN TOP ← 0;
  ELSE TOP ← TOP + 1;
FI;
```

### FPU Flags Affected

The C1 flag is set to 0; otherwise, cleared to 0. The C0, C2, and C3 flags are undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM EM or TS in CR0 is set.

## FINIT/FNINIT—Initialize Floating-Point Unit

Opcode	Instruction	Description
9B DB E3	FINIT	Initialize FPU after checking for pending unmasked floating-point exceptions.
DB E3	FNINIT*	Initialize FPU without checking for pending unmasked floating-point exceptions.

### NOTE:

\* See “IA-32 Architecture Compatibility” 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.

### 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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on a Pentium Pro 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;

```

## **FINIT/FNINIT—Initialize Floating-Point Unit (Continued)**

### **FPU Flags Affected**

C0, C1, C2, C3 cleared to 0.

### **Floating-Point Exceptions**

None.

### **Protected Mode Exceptions**

#NM                    EM or TS in CR0 is set.

### **Real-Address Mode Exceptions**

#NM                    EM or TS in CR0 is set.

### **Virtual-8086 Mode Exceptions**

#NM                    EM or TS in CR0 is set.

## FIST/FISTP—Store Integer

Opcode	Instruction	Description
DF /2	FIST <i>m16int</i>	Store ST(0) in <i>m16int</i>
DB /2	FIST <i>m32int</i>	Store ST(0) in <i>m32int</i>
DF /3	FISTP <i>m16int</i>	Store ST(0) in <i>m16int</i> and pop register stack
DB /3	FISTP <i>m32int</i>	Store ST(0) in <i>m32int</i> and pop register stack
DF /7	FISTP <i>m64int</i>	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 can also store values in quadword-integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

ST(0)	DEST
$-\infty$	*
$-F < -1$	-I
$-1 < -F < -0$	**
-0	0
+0	0
$+0 < +F < +1$	**
$+F > +1$	+I
$+\infty$	*
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.

## FIST/FISTP—Store Integer (Continued)

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 value being stored is too large for the destination format, is an  $\infty$ , is a NaN, or is in an unsupported format and if the invalid-arithmic-operand exception (#IA) is unmasked, an invalid-operation exception 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 the destination operand.

### 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.
	Cleared to 0 otherwise.
C0, C2, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is too large for the destination format
	Source operand is a NaN value or unsupported format.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable 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	EM or TS in CR0 is set.

## FIST/FISTP—Store Integer (Continued)

#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FLD—Load Floating Point Value

Opcode	Instruction	Description
D9 /0	FLD <i>m32fp</i>	Push <i>m32fp</i> onto the FPU register stack.
DD /0	FLD <i>m64fp</i>	Push <i>m64fp</i> onto the FPU register stack.
DB /5	FLD <i>m80fp</i>	Push <i>m80fp</i> onto the FPU register stack.
D9 C0+i	FLD ST(i)	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.

### Operation

```

IF SRC is ST(i)
  THEN
    temp ← ST(i)
FI;
TOP ← TOP – 1;
IF SRC is memory-operand
  THEN
    ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
  ELSE (* SRC is ST(i) *)
    ST(0) ← temp;
FI;

```

### FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, cleared to 0.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack overflow occurred.  
#IA Source operand is an SNaN value or unsupported format.  
#D Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.



## FLD—Load Floating Point Value (Continued)

### Protected Mode Exceptions

#GP(0)	If destination is located in a nonwritable 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

Opcode	Instruction	Description
D9 E8	FLD1	Push +1.0 onto the FPU register stack.
D9 E9	FLDL2T	Push $\log_2 10$ onto the FPU register stack.
D9 EA	FLDL2E	Push $\log_2 e$ onto the FPU register stack.
D9 EB	FLDPI	Push $\pi$ onto the FPU register stack.
D9 EC	FLDLG2	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Push $\log_8 2$ onto the FPU register stack.
D9 EE	FLDZ	Push +0.0 onto the FPU register stack.

### 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  $\log_8 2$ . 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.

See the section titled “Pi” in Chapter 8 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a description of the  $\pi$  constant.

### Operation

TOP  $\leftarrow$  TOP – 1;  
ST(0)  $\leftarrow$  CONSTANT;

### FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, cleared to 0.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack overflow occurred.

### Protected Mode Exceptions

#NM EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

## **FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant (Continued)**

### **Virtual-8086 Mode Exceptions**

#NM                      EM or TS in CR0 is set.

### **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.

## FLDCW—Load x87 FPU Control Word

Opcode	Instruction	Description
D9 /5	FLDCW <i>m2byte</i>	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 unmask 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 *IA-32 Intel Architecture 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.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## FLDCW—Load x87 FPU Control Word (Continued)

### 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	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FLDENV—Load x87 FPU Environment

Opcode	Instruction	Description
D9 /4	FLDENV <i>m14/28byte</i>	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 *IA-32 Intel Architecture 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 *IA-32 Intel Architecture 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.

### 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.

## FLDENV—Load x87 FPU Environment (Continued)

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FMUL/FMULP/FIMUL—Multiply

Opcode	Instruction	Description
D8 /1	FMUL <i>m32fp</i>	Multiply ST(0) by <i>m32fp</i> and store result in ST(0)
DC /1	FMUL <i>m64fp</i>	Multiply ST(0) by <i>m64fp</i> and store result in ST(0)
D8 C8+i	FMUL ST(0), ST(i)	Multiply ST(0) by ST(i) and store result in ST(0)
DC C8+i	FMUL ST(i), ST(0)	Multiply ST(i) by ST(0) and store result in ST(i)
DE C8+i	FMULP ST(i), ST(0)	Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack
DE C9	FMULP	Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack
DA /1	FIMUL <i>m32int</i>	Multiply ST(0) by <i>m32int</i> and store result in ST(0)
DE /1	FIMUL <i>m16int</i>	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.



## FMUL/FMULP/FIMUL—Multiply (Continued)

		DEST						
		$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
SRC	$-\infty$	$+\infty$	$+\infty$	*	*	$-\infty$	$-\infty$	NaN
	-F	$+\infty$	+F	+0	-0	-F	$-\infty$	NaN
	-I	$+\infty$	+F	+0	-0	-F	$-\infty$	NaN
	-0	*	+0	+0	-0	-0	*	NaN
	+0	*	-0	-0	+0	+0	*	NaN
	+I	$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
	+F	$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
	$+\infty$	$-\infty$	$-\infty$	*	*	$+\infty$	$+\infty$	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.

### Operation

IF instruction is FIMUL

THEN

DEST  $\leftarrow$  DEST \* ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (\* source operand is floating-point value \*)

DEST  $\leftarrow$  DEST \* SRC;

FI;

IF instruction  $\leftarrow$  FMULP

THEN

PopRegisterStack

FI;

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) fault is generated: 0  $\leftarrow$  not roundup; 1  $\leftarrow$  roundup.

C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

**FMUL/FMULP/FIMUL—Multiply (Continued)**

#IA	Operand is an SNaN value or unsupported format. One operand is $\pm 0$ and the other is $\pm \infty$ .
#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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

**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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FNOP—No Operation

Opcode	Instruction	Description
D9 D0	FNOP	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.

### FPU Flags Affected

C0, C1, C2, C3 undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM EM or TS in CR0 is set.

## FPATAN—Partial Arctangent

Opcode	Instruction	Description
D9 F3	FPATAN	Replace ST(1) with $\arctan(\text{ST}(1)/\text{ST}(0))$ and pop the register stack

### 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.

		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	$-\pi$	$-\pi$ to $-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$ to -0	-0	NaN
	-0	$-\pi$	$-\pi$	$-\pi^*$	$-0^*$	-0	-0	NaN
	+0	$+\pi$	$+\pi$	$+\pi^*$	$+0^*$	+0	+0	NaN
	+F	$+\pi$	$+\pi$ 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 *IA-32 Intel Architecture 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,  $\arctangent(0,0)$  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.

## FPATAN—Partial Arctangent (Continued)

### 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.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
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	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------



## FPREM—Partial Remainder

Opcode	Instruction	Description
D9 F8	FPREM	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:

$$\text{Remainder} \leftarrow \text{ST}(0) - (Q * \text{ST}(1))$$

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.

		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 ∞, 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.

## FPREM—Partial Remainder (Continued)

The FPREM instruction gets its name “partial remainder” because of the way it computes the remainder. This instructions 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.

### 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.

**FPREM—Partial Remainder (Continued)**

- #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 EM or TS in CR0 is set.

**Real-Address Mode Exceptions**

- #NM EM or TS in CR0 is set.

**Virtual-8086 Mode Exceptions**

- #NM EM or TS in CR0 is set.



## FPREM1—Partial Remainder

Opcode	Instruction	Description
D9 F5	FPREM1	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:

$$\text{Remainder} \leftarrow \text{ST}(0) - (Q * \text{ST}(1))$$

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.

		ST(1)						
		$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
ST(0)	$-\infty$	*	*	*	*	*	*	NaN
	-F	ST(0)	$\pm F$ or -0	**	**	$\pm F$ or -0	ST(0)	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	$\pm F$ or +0	**	**	$\pm F$ or +0	ST(0)	NaN
	$+\infty$	*	*	*	*	*	*	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).

## FPREM1—Partial Remainder (Continued)

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, the 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.

### 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.

## FPREM1—Partial Remainder (Continued)

### 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	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## FPTAN—Partial Tangent

Opcode	Instruction	Clocks	Description
D9 F2	FPTAN	17-173	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.

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 *IA-32 Intel Architecture 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.

## FPTAN—Partial Tangent (Continued)

### Operation

```

IF ST(0) < 263
THEN
    C2 ← 0;
    ST(0) ← tan(ST(0));
    TOP ← TOP – 1;
    ST(0) ← 1.0;
ELSE (*source operand is out-of-range *)
    C2 ← 1;
FI;

```

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C2	Set to 1 if source operand is outside the range $-2^{63}$ to $+2^{63}$ ; otherwise, cleared 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.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

**FRNDINT—Round to Integer**

Opcode	Instruction	Description
D9 FC	FRNDINT	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.

**Operation**

ST(0)  $\leftarrow$  RoundToIntegralValue(ST(0));

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 $\leftarrow$ not roundup; 1 $\leftarrow$ roundup.
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	EM or TS in CR0 is set.
-----	-------------------------

**Real-Address Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

**Virtual-8086 Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

## FRSTOR—Restore x87 FPU State

Opcode	Instruction	Description
DD /4	FRSTOR <i>m94/108byte</i>	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 *IA-32 Intel Architecture 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 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.

### 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.

**FRSTOR—Restore x87 FPU State (Continued)****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.  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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

**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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.



## FSAVE/FNSAVE—Store x87 FPU State

Opcode	Instruction	Description
9B DD /6	FSAVE <i>m94/108byte</i>	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>	Store FPU environment to <i>m94byte</i> or <i>m108byte</i> without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

### NOTE:

\* See “IA-32 Architecture Compatibility” 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 *IA-32 Intel Architecture 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 in 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

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 insure that the storage operation has been completed.

## FSAVE/FNSAVE—Store x87 FPU State (Continued)

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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on a Pentium Pro processor.

### Operation

(\* Save FPU State and Registers \*)

DEST[FPUControlWord] ← FPUControlWord;

DEST[FPUStatusWord] ← FPUStatusWord;

DEST[FPUTagWord] ← FPUTagWord;

DEST[FPUDataPointer] ← FPUDataPointer;

DEST[FPUInstructionPointer] ← FPUInstructionPointer;

DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;

DEST[ST(0)] ← ST(0);

DEST[ST(1)] ← ST(1);

DEST[ST(2)] ← ST(2);

DEST[ST(3)] ← ST(3);

DEST[ST(4)] ← ST(4);

DEST[ST(5)] ← ST(5);

DEST[ST(6)] ← ST(6);

DEST[ST(7)] ← ST(7);

(\* Initialize FPU \*)

FPUControlWord ← 037FH;

FPUStatusWord ← 0;

FPUTagWord ← FFFFH;

FPUDataPointer ← 0;

FPUInstructionPointer ← 0;

FPULastInstructionOpcode ← 0;

### FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0) If destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

## FSAVE/FNSAVE—Store x87 FPU State (Continued)

	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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

## 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.



### FSCALE—Scale

Opcode	Instruction	Description
D9 FD	FSCALE	Scale ST(0) by ST(1).

#### Description

Multiplies the destination operand by 2 to the power of the source operand and stores the result in the destination operand. The destination operand is a floating-point value that is located in register ST(0). The source operand is the nearest integer value that is smaller than the value in the ST(1) register (that is, the value in register ST(1) is truncated toward 0 to its nearest integer value to form the source operand). This instruction provides rapid multiplication or division by integral powers of 2 because it is implemented by simply adding an integer value (the source operand) to the exponent of the value in register ST(0). The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

		ST(1)		
		-N	0	+N
ST(0)	-∞	-∞	-∞	-∞
	-F	-F	-F	-F
	-0	-0	-0	-0
	+0	+0	+0	+0
	+F	+F	+F	+F
	+∞	+∞	+∞	+∞
	NaN	NaN	NaN	NaN

#### NOTES:

- F Means finite floating-point value.
- N Means integer.

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) ;
```

## FSCALE—Scale (Continued)

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.

### Operation

$$ST(0) \leftarrow ST(0) * 2^{ST(1)}$$

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
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	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## FSIN—Sine

Opcode	Instruction	Description
D9 FE	FSIN	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.

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
-F	-1 to +1
-0	-0
+0	+0
+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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

### Operation

```
IF ST(0) < 263
THEN
    C2 ← 0;
    ST(0) ← sin(ST(0));
ELSE (* source operand out of range *)
    C2 ← 1;
FI;
```

## FSIN—Sine (Continued)

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C2	Set to 1 if source operand is outside the range $-2^{63}$ to $+2^{63}$ ; otherwise, cleared 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	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## FSINCOS—Sine and Cosine

Opcode	Instruction	Description
D9 FB	FSINCOS	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.

SRC	DEST	
	ST(1) Cosine	ST(0) Sine
$-\infty$	*	*
-F	-1 to +1	-1 to +1
-0	+1	-0
+0	+1	+0
+F	-1 to +1	-1 to +1
$+\infty$	*	*
NaN	NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range  $-2^{63}$  to  $+2^{63}$  can be reduced to the range of the instruction by subtracting an appropriate integer multiple of  $2\pi$  or by using the FPREM instruction with a divisor of  $2\pi$ . See the section titled “Pi” in Chapter 8 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for  $\pi$  in performing such reductions.



## FSINCOS—Sine and Cosine (Continued)

### Operation

```

IF ST(0) < 263
THEN
    C2 ← 0;
    TEMP ← cosine(ST(0));
    ST(0) ← sine(ST(0));

    TOP ← TOP - 1;
    ST(0) ← TEMP;
ELSE (* source operand out of range *)
    C2 ← 1;
FI:

```

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C2	Set to 1 if source operand is outside the range $-2^{63}$ to $+2^{63}$ ; otherwise, cleared 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.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## FSQRT—Square Root

Opcode	Instruction	Description
D9 FA	FSQRT	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.

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.

### Operation

ST(0)  $\leftarrow$  SquareRoot(ST(0));

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if inexact-result exception (#P) is generated: 0 $\leftarrow$ not roundup; 1 $\leftarrow$ roundup.
C0, C2, C3	Undefined.

## FSQRT—Square Root (Continued)

### 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	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## FST/FSTP—Store Floating Point Value

Opcode	Instruction	Description
D9 /2	FST <i>m32fp</i>	Copy ST(0) to <i>m32fp</i>
DD /2	FST <i>m64fp</i>	Copy ST(0) to <i>m64fp</i>
DD D0+i	FST ST(i)	Copy ST(0) to ST(i)
D9 /3	FSTP <i>m32fp</i>	Copy ST(0) to <i>m32fp</i> and pop register stack
DD /3	FSTP <i>m64fp</i>	Copy ST(0) to <i>m64fp</i> and pop register stack
DB /7	FSTP <i>m80fp</i>	Copy ST(0) to <i>m80fp</i> and pop register stack
DD D8+i	FSTP ST(i)	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 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.

### Operation

DEST  $\leftarrow$  ST(0);

IF instruction  $\leftarrow$  FSTP

THEN

PopRegisterStack; FI;

## FST/FSTP—Store Floating Point Value (Continued)

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C0, C2, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported 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 nonwritable 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

**FST/FSTP—Store Floating Point Value (Continued)****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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FSTCW/FNSTCW—Store x87 FPU Control Word

Opcode	Instruction	Description
9B D9 /7	FSTCW <i>m2byte</i>	Store FPU control word to <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW* <i>m2byte</i>	Store FPU control word to <i>m2byte</i> without checking for pending unmasked floating-point exceptions.

### NOTE:

\* See “IA-32 Architecture Compatibility” 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.

### 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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on a Pentium Pro processor.

### Operation

DEST ← FPUControlWord;

### FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

### Floating-Point Exceptions

None.

## FSTCW/FNSTCW—Store x87 FPU Control Word (Continued)

### Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.



## FSTENV/FNSTENV—Store x87 FPU Environment

Opcode	Instruction	Description
9B D9 /6	FSTENV <i>m14/28byte</i>	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>	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.

### NOTE:

\* See “IA-32 Architecture Compatibility” 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 *IA-32 Intel Architecture 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 in 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 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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on a Pentium Pro processor.

## FSTENV/FNSTENV—Store x87 FPU Environment (Continued)

### Operation

DEST[FPUControlWord] ← FPUControlWord;  
 DEST[FPUStatusWord] ← FPUStatusWord;  
 DEST[FPUTagWord] ← FPUTagWord;  
 DEST[FPUDataPointer] ← FPUDataPointer;  
 DEST[FPUInstructionPointer] ← FPUInstructionPointer;  
 DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;

### FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a nonwritable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

## FSTENV/FNSTENV—Store x87 FPU Environment (Continued)

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FSTSW/FNSTSW—Store x87 FPU Status Word

Opcode	Instruction	Description
9B DD /7	FSTSW <i>m2byte</i>	Store FPU status word at <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW* <i>m2byte</i>	Store FPU status word at <i>m2byte</i> without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW* AX	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

### NOTE:

\* See “IA-32 Architecture Compatibility” 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 *IA-32 Intel Architecture 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 in 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 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 *IA-32 Intel Architecture 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 or P6 family processor.

## FSTSW/FNSTSW—Store x87 FPU Status Word (Continued)

### 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 nonwritable 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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

### 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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.

## **FSTSW/FNSTSW—Store x87 FPU Status Word (Continued)**

#AC(0)                      If alignment checking is enabled and an unaligned memory reference is made.

## FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	Description
D8 /4	FSUB <i>m32fp</i>	Subtract <i>m32fp</i> from ST(0) and store result in ST(0)
DC /4	FSUB <i>m64fp</i>	Subtract <i>m64fp</i> from ST(0) and store result in ST(0)
D8 E0+i	FSUB ST(0), ST(i)	Subtract ST(i) from ST(0) and store result in ST(0)
DC E8+i	FSUB ST(i), ST(0)	Subtract ST(0) from ST(i) and store result in ST(i)
DE E8+i	FSUBP ST(i), ST(0)	Subtract ST(0) from ST(i), store result in ST(i), and pop register stack
DE E9	FSUBP	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack
DA /4	FISUB <i>m32int</i>	Subtract <i>m32int</i> from ST(0) and store result in ST(0)
DE /4	FISUB <i>m16int</i>	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.

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 SRC value is subtracted from the DEST value ( $DEST - SRC \leftarrow \text{result}$ ).

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  $+0 - (-0) \leftarrow +0$ , and that  $-0 - (+0) \leftarrow -0$ . 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.

## FSUB/FSUBP/FISUB—Subtract (Continued)

		SRC						
		$-\infty$	$-F$ or $-I$	$-0$	$+0$	$+F$ or $+I$	$+\infty$	NaN
DEST	$-\infty$	*	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN
	$-F$	$+\infty$	$\pm F$ or $\pm 0$	DEST	DEST	$-F$	$-\infty$	NaN
	$-0$	$+\infty$	$-SRC$	$\pm 0$	$-0$	$-SRC$	$-\infty$	NaN
	$+0$	$+\infty$	$-SRC$	$+0$	$\pm 0$	$-SRC$	$-\infty$	NaN
	$+F$	$+\infty$	$+F$	DEST	DEST	$\pm F$ or $\pm 0$	$-\infty$	NaN
	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

F Means finite floating-point value.

I Means integer.

\* Indicates floating-point invalid-arithmatic-operand (#IA) exception.

**Operation**

IF instruction is FISUB

THEN

DEST  $\leftarrow$  DEST – ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (\* source operand is floating-point value \*)

DEST  $\leftarrow$  DEST – SRC;

FI;

IF instruction is FSUBP

THEN

PopRegisterStack

FI;

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) fault is generated: 0  $\leftarrow$  not roundup; 1  $\leftarrow$  roundup.

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.



**FSUB/FSUBP/FISUB—Subtract (Continued)**

#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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

**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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	Description
D8 /5	FSUBR <i>m32fp</i>	Subtract ST(0) from <i>m32fp</i> and store result in ST(0)
DC /5	FSUBR <i>m64fp</i>	Subtract ST(0) from <i>m64fp</i> and store result in ST(0)
D8 E8+i	FSUBR ST(0), ST(i)	Subtract ST(0) from ST(i) and store result in ST(0)
DC E0+i	FSUBR ST(i), ST(0)	Subtract ST(i) from ST(0) and store result in ST(i)
DE E0+i	FSUBRP ST(i), ST(0)	Subtract ST(i) from ST(0), store result in ST(i), and pop register stack
DE E1	FSUBRP	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack
DA /5	FISUBR <i>m32int</i>	Subtract ST(0) from <i>m32int</i> and store result in ST(0)
DE /5	FISUBR <i>m16int</i>	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 ( $SRC - DEST \leftarrow \text{result}$ ).

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  $+0 - (-0) \leftarrow +0$ , and that  $-0 - (+0) \leftarrow -0$ . 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.

## FSUBR/FSUBRP/FISUBR—Reverse Subtract (Continued)

		SRC						
		$-\infty$	$-F$ or $-I$	$-0$	$+0$	$+F$ or $+I$	$+\infty$	NaN
DEST	$-\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	$-F$	$-\infty$	$\pm F$ or $\pm I$	$-DEST$	$-DEST$	$+F$	$+\infty$	NaN
	$-0$	$-\infty$	SRC	$\pm 0$	$+0$	SRC	$+\infty$	NaN
	$+0$	$-\infty$	SRC	$-0$	$\pm 0$	SRC	$+\infty$	NaN
	$+F$	$-\infty$	$-F$	$-DEST$	$-DEST$	$\pm F$ or $\pm I$	$+\infty$	NaN
	$+\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	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.

### Operation

IF instruction is FISUBR

THEN

DEST  $\leftarrow$  ConvertToDoubleExtendedPrecisionFP(SRC)  $-$  DEST;

ELSE (\* source operand is floating-point value \*)

DEST  $\leftarrow$  SRC  $-$  DEST;

FI;

IF instruction  $\leftarrow$  FSUBRP

THEN

PopRegisterStack

FI;

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) fault is generated: 0  $\leftarrow$  not roundup; 1  $\leftarrow$  roundup.

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.

**FSUBR/FSUBRP/FISUBR—Reverse Subtract (Continued)**

#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	EM or TS in CR0 is set.
#PF(fault-code)	If a 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.
#NM	EM or TS in CR0 is set.

**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	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## FTST—TEST

Opcode	Instruction	Description
D9 E4	FTST	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).

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” 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  $-0.0 \leftarrow +0.0$ .

### Operation

CASE (relation of operands) OF

Not comparable: C3, C2, C0 ← 111;

ST(0) > 0.0: C3, C2, C0 ← 000;

ST(0) < 0.0: C3, C2, C0 ← 001;

ST(0) ← 0.0: C3, C2, C0 ← 100;

ESAC;

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred; otherwise, cleared to 0.

C0, C2, C3 See above table.

### 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.

## FTST—TEST (Continued)

### Protected Mode Exceptions

#NM                      EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM                      EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM                      EM or TS in CR0 is set.

## FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values

<b>Opcode</b>	<b>Instruction</b>	<b>Description</b>
DD E0+i	FUCOM ST(i)	Compare ST(0) with ST(i)
DD E1	FUCOM	Compare ST(0) with ST(1)
DD E8+i	FUCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack
DD E9	FUCOMP	Compare ST(0) with ST(1) and pop register stack
DA E9	FUCOMPP	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  $-0.0 \leftarrow +0.0$ .

<b>Comparison Results</b>	<b>C3</b>	<b>C2</b>	<b>C0</b>
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 $\leftarrow$ ST(i)	1	0	0
Unordered	1	1	1

#### NOTE:

\* 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” in this chapter). The FUCOM instructions perform the same operations as the FCOM instructions. The only difference is that the FUCOM 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 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 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.

## FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values (Continued)

### Operation

CASE (relation of operands) OF

ST > SRC: C3, C2, C0 ← 000;

ST < SRC: C3, C2, C0 ← 001;

ST ← SRC: C3, C2, C0 ← 100;

ESAC;

IF ST(0) or SRC ← QNaN, but not SNaN or unsupported format

THEN

C3, C2, C0 ← 111;

ELSE (\* ST(0) or SRC is SNaN or unsupported format \*)

#IA;

IF FPUControlWord.IM ← 1

THEN

C3, C2, C0 ← 111;

FI;

FI;

IF instruction ← FUCOMP

THEN

PopRegisterStack;

FI;

IF instruction ← FUCOMPP

THEN

PopRegisterStack;

PopRegisterStack;

FI;

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

C0, C2, C3 See table on previous page.

### 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 EM or TS in CR0 is set.



## FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values (Continued)

### Real-Address Mode Exceptions

#NM                    EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM                    EM or TS in CR0 is set.

## **FWAIT—Wait**

See entry for WAIT/FWAIT—Wait.

## FXAM—Examine

Opcode	Instruction	Description
D9 E5	FXAM	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).

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.

### Operation

C1 ← sign bit of ST; (\* 0 for positive, 1 for negative \*)

CASE (class of value or number in ST(0)) OF

  Unsupported: C3, C2, C0 ← 000;

  NaN: C3, C2, C0 ← 001;

  Normal: C3, C2, C0 ← 010;

  Infinity: C3, C2, C0 ← 011;

  Zero: C3, C2, C0 ← 100;

  Empty: C3, C2, C0 ← 101;

  Denormal: C3, C2, C0 ← 110;

ESAC;

### FPU Flags Affected

C1 Sign of value in ST(0).

C0, C2, C3 See table above.

## **FXAM—Examine (Continued)**

### **Floating-Point Exceptions**

None.

### **Protected Mode Exceptions**

#NM                    EM or TS in CR0 is set.

### **Real-Address Mode Exceptions**

#NM                    EM or TS in CR0 is set.

### **Virtual-8086 Mode Exceptions**

#NM                    EM or TS in CR0 is set.

## FXCH—Exchange Register Contents

Opcode	Instruction	Description
D9 C8+i	FXCH ST(i)	Exchange the contents of ST(0) and ST(i)
D9 C9	FXCH	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);
```

### Operation

IF number-of-operands is 1

THEN

temp  $\leftarrow$  ST(0);

ST(0)  $\leftarrow$  SRC;

SRC  $\leftarrow$  temp;

ELSE

temp  $\leftarrow$  ST(0);

ST(0)  $\leftarrow$  ST(1);

ST(1)  $\leftarrow$  temp;

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred; otherwise, cleared to 0.

C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

### Protected Mode Exceptions

#NM EM or TS in CR0 is set.

## **FXCH—Exchange Register Contents (Continued)**

### **Real-Address Mode Exceptions**

#NM                      EM or TS in CR0 is set.

### **Virtual-8086 Mode Exceptions**

#NM                      EM or TS in CR0 is set.

## FXRSTOR—Restore x87 FPU, MMX, SSE, and SSE2 State

Opcode	Instruction	Description
0F AE /1	FXRSTOR <i>m512byte</i>	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .

### Description

Reloads the x87 FPU, MMX, 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 the first byte of the data should be located on a 16-byte boundary. Table 3-13 shows the layout of the state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions.

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as that shown in Table 3-13. Referencing a state image saved with an FSAVE or FNSAVE instruction 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.

Bit 6 and 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.

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment. (See alignment check exception [#AC] below.)

## FXRSTOR—Restore x87 FPU, MMX, SSE, and SSE2 State (Continued)

#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If CPUID feature flag FXSR is 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).

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If CPUID feature flag SSE2 is 0. If instruction is preceded by a LOCK override prefix.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.



## FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State

Opcode	Instruction	Description
OF AE /0	FXSAVE <i>m512byte</i>	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m512byte</i> .

### Description

Saves the current state of the x87 FPU, MMX, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. Table 3-13 shows the layout of the state information in memory.

**Table 3-13. Layout of FXSAVE and FXRSTOR Memory Region**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Rsrvd		CS		FPU IP				FOP		FTW		FSW		FCW		0
MXCSR_MASK				MXCSR				Rsrvd	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		
								Reserved						288		
								Reserved						304		
								Reserved						320		
								Reserved						336		
								Reserved						352		
								Reserved						368		
								Reserved						384		
								Reserved						400		
								Reserved						416		
								Reserved						432		
								Reserved						448		
								Reserved						464		
								Reserved						480		
								Reserved						496		

## FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State (Continued)

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, and/or XMM and MXCSR registers.

The fields in Table 3-13 are as follows:

FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the <i>IA-32 Intel Architecture 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>IA-32 Intel Architecture Software Developer's Manual, Volume 1</i> , for the layout of the x87 FPU status word.
FTW	x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs. See Figure 8-7 in the <i>IA-32 Intel Architecture Software Developer's Manual, Volume 1</i> , for the layout of the x87 FPU tag word.
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>IA-32 Intel Architecture 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: <ul style="list-style-type: none"><li>• 32-bit mode—32-bit IP offset.</li><li>• 16-bit mode—low 16 bits are IP offset; high 16 bits are reserved.</li></ul> See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>IA-32 Intel Architecture Software Developer's Manual, Volume 1</i> , for a description of the x87 FPU instruction pointer.
CS	x87 FPU Instruction Pointer Selector (16 bits).

## FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State (Continued)

FPU DP	<p>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:</p> <ul style="list-style-type: none"> <li>• 32-bit mode—32-bit IP offset.</li> <li>• 16-bit mode—low 16 bits are IP offset; high 16 bits are reserved.</li> </ul> <p>See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>IA-32 Intel Architecture Software Developer’s Manual, Volume 1</i>, for a description of the x87 FPU operand pointer.</p>
DS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits).
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the <i>IA-32 Intel Architecture 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 value can be used to adjust a value to be written to the MXCSR register to ensure that all reserved bits are set to 0. Setting the reserved bits to 0 prevents a general-protection exception (#GP) from being generated when writing to the MXCSR register with an FXRSTOR or LDMXCSR instruction. See “Guidelines for Writing to the MXCSR Register” in Chapter 11 of the <i>IA-32 Intel Architecture 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 registers. These 80-bit fields contain the x87 FPU data registers or the MMX 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 registers), the MMX registers are saved. When the MMX 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.

## FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State (Continued)

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, 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 a 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.

Note that 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 registers) using the following table:

## FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State (Continued)

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.

### Operation

DEST ← Save(x87 FPU, MMX, XMM7-XMM0, MXCSR);

### Protected Mode Exceptions

- #GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  
If 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 TS in CR0 is set.

**FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State (Continued)**

#UD	<p>If EM in CR0 is set.</p> <p>If CPUID feature flag FXSR is 0.</p> <p>If instruction is preceded by a LOCK override prefix.</p>
#AC	<p>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).</p>

**Real-Address Mode Exceptions**

#GP(0)	<p>If memory operand is not aligned on a 16-byte boundary, regardless of segment.</p>
Interrupt 13	<p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p>
#NM	<p>If TS in CR0 is set.</p>
#UD	<p>If EM in CR0 is set.</p> <p>If CPUID feature flag FXSR is 0.</p> <p>If instruction is preceded by a LOCK override prefix.</p>

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	<p>For a page fault.</p>
#AC	<p>For unaligned memory reference if the current privilege level is 3.</p>

**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 *IA-32 Intel Architecture Software Developer's Manual, Volume 3*. This order vary for the FXSAVE instruction for different IA-32 processor implementations.

## FXTRACT—Extract Exponent and Significand

Opcode	Instruction	Description
D9 F4	FXTRACT	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  $\log_b(x)$  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).

### 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 occurred.  
                          Stack overflow occurred.  
#IA                    Source operand is an SNaN value or unsupported format.

**FXTRACT—Extract Exponent and Significand (Continued)**

#Z ST(0) operand is  $\pm 0$ .

#D Source operand is a denormal value.

**Protected Mode Exceptions**

#NM EM or TS in CR0 is set.

**Real-Address Mode Exceptions**

#NM EM or TS in CR0 is set.

**Virtual-8086 Mode Exceptions**

#NM EM or TS in CR0 is set.



## FYL2X—Compute $y * \log_2 x$

Opcode	Instruction	Description
D9 F1	FYL2X	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.

		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):

$$\log_b x \leftarrow (\log_2 b)^{-1} * \log_2 x$$

### Operation

$ST(1) \leftarrow ST(1) * \log_2 ST(0);$

PopRegisterStack;

**FYL2X—Compute  $y * \log_2 x$  (Continued)****FPU Flags Affected**

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
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 $\pm 0$ .
#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	EM or TS in CR0 is set.
-----	-------------------------

**Real-Address Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

**Virtual-8086 Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

## FYL2XP1—Compute $y * \log_2(x + 1)$

Opcode	Instruction	Description
D9 F9	FYL2XP1	Replace ST(1) with $ST(1) * \log_2(ST(0) + 1.0)$ and pop the register stack

### Description

Computes the log epsilon ( $ST(1) * \log_2(ST(0) + 1.0)$ ), 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.

		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:

$$\text{scale factor} \leftarrow \log_n 2$$

**FYL2XP1—Compute  $y * \log_2(x + 1)$  (Continued)****Operation**

$ST(1) \leftarrow ST(1) * \log_2(ST(0) + 1.0);$   
 PopRegisterStack;

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
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	EM or TS in CR0 is set.
-----	-------------------------

**Real-Address Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

**Virtual-8086 Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

## HLT—Halt

Opcode	Instruction	Description
F4	HLT	Halt

### 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.

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.

### Operation

Enter Halt state;

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

#GP(0) If the current privilege level is not 0.

## IDIV—Signed Divide

Opcode	Instruction	Description
F6 /7	IDIV <i>r/m8</i>	Signed divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder
F7 /7	IDIV <i>r/m16</i>	Signed divide DX:AX by <i>r/m16</i> , with result stored in AX ← Quotient, DX ← Remainder
F7 /7	IDIV <i>r/m32</i>	Signed divide EDX:EAX by <i>r/m32</i> , with result stored in EAX ← Quotient, EDX ← Remainder

### Description

Divides (signed) the value in the AX, DX:AX, or EDX:EAX registers (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), as shown in the following table:

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	<i>r/m8</i>	AL	AH	−128 to +127
Doubleword/word	DX:AX	<i>r/m16</i>	AX	DX	−32,768 to +32,767
Quadword/doubleword	EDX:EAX	<i>r/m32</i>	EAX	EDX	−2 <sup>31</sup> to 2 <sup>32</sup> − 1

Non-integral results are truncated (chopped) towards 0. The sign of the remainder is always the same as the sign of the dividend. The absolute value of the remainder is always less than the absolute value of the divisor. Overflow is indicated with the #DE (divide error) exception rather than with the OF (overflow) flag.

### Operation

```

IF SRC ← 0
  THEN #DE; (* divide error *)
FI;
IF OpernadSize ← 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;
  
```

**IDIV—Signed Divide (Continued)**

ELSE

IF OpernadSize  $\leftarrow$  16 (\* doubleword/word operation \*)

THEN

temp  $\leftarrow$  DX:AX / SRC; (\* signed division \*)

IF (temp &gt; 7FFFH) OR (temp &lt; 8000H)

(\* if a positive result is greater than 7FFFH \*)

(\* or a negative result is less than 8000H \*)

THEN #DE; (\* divide error \*) ;

ELSE

AX  $\leftarrow$  temp;DX  $\leftarrow$  DX:AX SignedModulus SRC;

FI;

ELSE (\* quadword/doubleword operation \*)

temp  $\leftarrow$  EDX:EAX / SRC; (\* signed division \*)

IF (temp &gt; 7FFFFFFFH) OR (temp &lt; 80000000H)

(\* if a positive result is greater than 7FFFFFFFH \*)

(\* or a negative result is less than 80000000H \*)

THEN #DE; (\* divide error \*) ;

ELSE

EAX  $\leftarrow$  temp;EDX  $\leftarrow$  EDX:EAX SignedModulus SRC;

FI;

FI;

FI;

**Flags Affected**

The CF, OF, SF, ZF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

#DE	If the source operand (divisor) is 0. The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## IDIV—Signed Divide (Continued)

### 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.

### 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.



## IMUL—Signed Multiply

Opcode	Instruction	Description
F6 /5	IMUL <i>r/m8</i>	AX ← AL * <i>r/m</i> byte
F7 /5	IMUL <i>r/m16</i>	DX:AX ← AX * <i>r/m</i> word
F7 /5	IMUL <i>r/m32</i>	EDX:EAX ← EAX * <i>r/m</i> doubleword
0F AF /r	IMUL <i>r16,r/m16</i>	word register ← word register * <i>r/m</i> word
0F AF /r	IMUL <i>r32,r/m32</i>	doubleword register ← doubleword register * <i>r/m</i> doubleword
6B /r ib	IMUL <i>r16,r/m16,imm8</i>	word register ← <i>r/m16</i> * sign-extended immediate byte
6B /r ib	IMUL <i>r32,r/m32,imm8</i>	doubleword register ← <i>r/m32</i> * sign-extended immediate byte
6B /r ib	IMUL <i>r16,imm8</i>	word register ← word register * sign-extended immediate byte
6B /r ib	IMUL <i>r32,imm8</i>	doubleword register ← doubleword register * sign-extended immediate byte
69 /r iw	IMUL <i>r16,r/m16,imm16</i>	word register ← <i>r/m16</i> * immediate word
69 /r id	IMUL <i>r32,r/m32,imm32</i>	doubleword register ← <i>r/m32</i> * immediate doubleword
69 /r iw	IMUL <i>r16,imm16</i>	word register ← <i>r/m16</i> * immediate word
69 /r id	IMUL <i>r32,imm32</i>	doubleword register ← <i>r/m32</i> * immediate doubleword

### 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, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX 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.

## IMUL—Signed Multiply (Continued)

The CF and OF flags are set when significant bits are carried into the upper half of the result. The CF and OF flags are cleared when the result 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, 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.

### Operation

```

IF (NumberOfOperands ← 1)
  THEN IF (OperandSize ← 8)
    THEN
      AX ← AL * SRC (* signed multiplication *)
      IF ((AH ← 00H) OR (AH ← FFH))
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1;
      FI;
    ELSE IF OperandSize ← 16
      THEN
        DX:AX ← AX * SRC (* signed multiplication *)
        IF ((DX ← 0000H) OR (DX ← FFFFH))
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1;
        FI;
      ELSE (* OperandSize ← 32 *)
        EDX:EAX ← EAX * SRC (* signed multiplication *)
        IF ((EDX ← 00000000H) OR (EDX ← FFFFFFFFH))
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1;
        FI;
      FI;
    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;
        FI;
      ELSE (* NumberOfOperands ← 3 *)

```

**IMUL—Signed Multiply (Continued)**

```

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;
FI;
FI;
FI;

```

**Flags Affected**

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If the DS, ES, FS, or GS register is used to access memory and it contains a null 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**

#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.

## IN—Input from Port

Opcode	Instruction	Description
E4 <i>ib</i>	IN AL, <i>imm8</i>	Input byte from <i>imm8</i> I/O port address into AL
E5 <i>ib</i>	IN AX, <i>imm8</i>	Input byte from <i>imm8</i> I/O port address into AX
E5 <i>ib</i>	IN EAX, <i>imm8</i>	Input byte from <i>imm8</i> I/O port address into EAX
EC	IN AL,DX	Input byte from I/O port in DX into AL
ED	IN AX,DX	Input word from I/O port in DX into AX
ED	IN EAX,DX	Input doubleword from I/O port in DX into EAX

### 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 12, *Input/Output*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

### 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; (* Reads from selected I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Reads from selected I/O port *)
  FI;
```

### Flags Affected

None.

## IN—Input from Port (Continued)

### 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.

### Real-Address Mode Exceptions

None.

### 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.

## INC—Increment by 1

Opcode	Instruction	Description
FE /0	INC <i>r/m8</i>	Increment <i>r/m</i> byte by 1
FF /0	INC <i>r/m16</i>	Increment <i>r/m</i> word by 1
FF /0	INC <i>r/m32</i>	Increment <i>r/m</i> doubleword by 1
40+ <i>rw</i>	INC <i>r16</i>	Increment word register by 1
40+ <i>rd</i>	INC <i>r32</i>	Increment doubleword register by 1

### 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.

### 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 nonwritable 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.

## INC—Increment by 1 (Continued)

### 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.

## INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Description
6C	INS m8, DX	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INS m16, DX	Input word from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INS m32, DX	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI
6C	INSB	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI
6D	INSW	Input word from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INSD	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI

### 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:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, 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 (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 (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.



## INS/INSB/INSW/INSD—Input from Port to String (Continued)

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 this chapter 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 12, *Input/Output*, in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

### 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; (* Reads from I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Reads from I/O port *)
  FI;
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;
FI;
```

### Flags Affected

None.

## INS/INSB/INSW/INSD—Input from Port to String (Continued)

### 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 nonwritable 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.

### 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 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.

## INT *n*/INTO/INT 3—Call to Interrupt Procedure

Opcode	Instruction	Description
CC	INT 3	Interrupt 3—trap to debugger
CD <i>ib</i>	INT <i>imm8</i>	Interrupt vector number specified by immediate byte
CE	INTO	Interrupt 4—if overflow flag is 1

### 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 *IA-32 Intel Architecture 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 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.

## INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)

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).

PE	0	1	1	1	1	1	1	1
VM	–	–	–	–	–	0	1	1
IOPL	–	–	–	–	–	–	<3	=3
DPL/CPL RELATIONSHIP	–	DPL< CPL	–	DPL> CPL	DPL= CPL or C	DPL< CPL & NC	–	–
INTERRUPT TYPE	–	S/W	–	–	–	–	–	–
GATE TYPE	–	–	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

### NOTES:

- Don't Care.
- Y Yes, Action Taken.
- Blank Action Not Taken.

## INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)

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 general protection exception (#GP); 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 three 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 and exceptions.

```

IF PE=0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE=1 *)
    IF (VM=1 AND IOPL < 3 AND INT n)
      THEN
        #GP(0);
      ELSE (* protected mode or virtual-8086 mode interrupt *)
        GOTO PROTECTED-MODE;
    FI;
  FI;

REAL-ADDRESS-MODE:
  IF ((DEST * 4) + 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 *)
  CS ← IDT(Descriptor (vector_number * 4), selector));
  EIP ← IDT(Descriptor (vector_number * 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;

PROTECTED-MODE:
  IF ((DEST * 8) + 7) is not within IDT limits
    OR selected IDT descriptor is not an interrupt-, trap-, or task-gate type
    THEN #GP((DEST * 8) + 2 + EXT);
    (* EXT is bit 0 in error code *)
  FI;

```

**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

```

IF software interrupt (* generated by INT n, INT 3, or INTO *)
  THEN
    IF gate descriptor DPL < CPL
      THEN #GP((vector_number * 8) + 2);
      (* PE=1, DPL<CPL, software interrupt *)
    FI;
  IF gate not present THEN #NP((vector_number * 8) + 2 + EXT); FI;
  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;

TASK-GATE: (* PE=1, task gate *)
  Read segment selector in task gate (IDT descriptor);
  IF local/global bit is set to local
    OR index not within GDT limits
      THEN #GP(TSS selector);
  FI;
  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);
  FI;
  IF TSS not present
    THEN #NP(TSS selector);
  FI;
  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(0);
      FI;
      Push(error code);
    FI;
  IF EIP not within code segment limit
    THEN #GP(0);
  FI;
END;
TRAP-OR-INTERRUPT-GATE
  Read segment selector for trap or interrupt gate (IDT descriptor);
  IF segment selector for code segment is null
    THEN #GP(0H + EXT); (* null selector with EXT flag set *)
  FI;

```

**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

```

IF segment selector is not within its descriptor table limits
    THEN #GP(selector + EXT);
FI;
Read trap or interrupt handler descriptor;
IF descriptor does not indicate a code segment
    OR code segment descriptor DPL > CPL
    THEN #GP(selector + EXT);
FI;
IF trap or interrupt gate segment is not present,
    THEN #NP(selector + EXT);
FI;
IF code segment is non-conforming AND DPL < CPL
    THEN IF VM=0
        THEN
            GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
            (* PE=1, interrupt or trap gate, nonconforming *)
            (* code segment, DPL<CPL, VM=0 *)
        ELSE (* VM=1 *)
            IF code segment DPL ≠ 0 THEN #GP(new code segment selector); FI;
            GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE;
            (* PE=1, interrupt or trap gate, DPL<CPL, VM=1 *)
        FI;
    ELSE (* PE=1, interrupt or trap gate, DPL ≥ CPL *)
        IF VM=1 THEN #GP(new code segment selector); FI;
        IF code segment is conforming OR code segment DPL ← CPL
            THEN
                GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
            ELSE
                #GP(CodeSegmentSelector + EXT);
                (* PE=1, interrupt or trap gate, nonconforming *)
                (* code segment, DPL>CPL *)
            FI;
    FI;
END;

INTER-PRIVILEGE-LEVEL-INTERRUPT
(* PE=1, interrupt or trap gate, non-conforming code segment, DPL<CPL *)
(* Check segment selector and descriptor for stack of new privilege level in current TSS *)
IF current TSS is 32-bit TSS
    THEN
        TSSstackAddress ← (new code segment DPL * 8) + 4
        IF (TSSstackAddress + 7) > TSS limit
            THEN #TS(current TSS selector); FI;
        NewSS ← TSSstackAddress + 4;
        NewESP ← stack address;

```

**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

```

ELSE (* TSS is 16-bit *)
    TSSstackAddress ← (new code segment DPL * 4) + 2
    IF (TSSstackAddress + 4) > TSS limit
        THEN #TS(current TSS selector); FI;
    NewESP ← TSSstackAddress;
    NewSS ← TSSstackAddress + 2;
FI;
IF segment selector is null THEN #TS(EXT); FI;
IF segment selector index is not within its descriptor table limits
    OR segment selector's RPL ≠ DPL of code segment,
    THEN #TS(SS selector + EXT);
FI;
Read segment descriptor for stack segment in GDT or LDT;
IF stack segment DPL ≠ DPL of code segment,
    OR stack segment does not indicate writable data segment,
    THEN #TS(SS selector + EXT);
FI;
IF stack segment not present THEN #SS(SS selector+EXT); FI;
IF 32-bit gate
    THEN
        IF new stack does not have room for 24 bytes (error code pushed)
            OR 20 bytes (no error code pushed)
            THEN #SS(segment selector + EXT);
        FI;
    ELSE (* 16-bit gate *)
        IF new stack does not have room for 12 bytes (error code pushed)
            OR 10 bytes (no error code pushed);
            THEN #SS(segment selector + EXT);
        FI;
    FI;
IF instruction pointer is not within code segment limits THEN #GP(0); FI;
SS:ESP ← TSS(NewSS:NewESP) (* segment descriptor information also loaded *)
IF 32-bit gate
    THEN
        CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
    ELSE (* 16-bit gate *)
        CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
    FI;
IF 32-bit gate
    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 *)

```



**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

```

ELSE(* 16-bit gate *)
    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 *)
FI;
CPL ← CodeSegmentDescriptor(DPL);
CS(RPL) ← CPL;
IF interrupt gate
    THEN IF ← 0 (* interrupt flag to 0 (disabled) *); FI;
TF ← 0;
VM ← 0;
RF ← 0;
NT ← 0;
END;

```

**INTERRUPT-FROM-VIRTUAL-8086-MODE:**

(\* Check segment selector and descriptor for privilege level 0 stack in current TSS \*)

IF current TSS is 32-bit TSS

THEN

TSSstackAddress ← (new code segment DPL \* 8) + 4

IF (TSSstackAddress + 7) > TSS limit

THEN #TS(current TSS selector); FI;

NewSS ← TSSstackAddress + 4;

NewESP ← stack address;

ELSE (\* TSS is 16-bit \*)

TSSstackAddress ← (new code segment DPL \* 4) + 2

IF (TSSstackAddress + 4) > TSS limit

THEN #TS(current TSS selector); FI;

NewESP ← TSSstackAddress;

NewSS ← TSSstackAddress + 2;

FI;

IF segment selector is null THEN #TS(EXT); FI;

IF segment selector index is not within its descriptor table limits

OR segment selector's RPL ≠ DPL of code segment,

THEN #TS(SS selector + EXT);

FI;

Access segment descriptor for stack segment in GDT or LDT;

IF stack segment DPL ≠ DPL of code segment,

OR stack segment does not indicate writable data segment,

THEN #TS(SS selector + EXT);

FI;

IF stack segment not present THEN #SS(SS selector+EXT); FI;

**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

```

IF 32-bit gate
  THEN
    IF new stack does not have room for 40 bytes (error code pushed)
      OR 36 bytes (no error code pushed);
      THEN #SS(segment selector + EXT);
    FI;
  ELSE (* 16-bit gate *)
    IF new stack does not have room for 20 bytes (error code pushed)
      OR 18 bytes (no error code pushed);
      THEN #SS(segment selector + EXT);
    FI;
  FI;
IF instruction pointer is not within code segment limits THEN #GP(0); FI;
tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
IF service through interrupt gate THEN IF ← 0; FI;
TempSS ← SS;
TempESP ← ESP;
SS:ESP ← TSS(SS0:ESP0); (* Change to level 0 stack segment *)
(* Following pushes are 16 bits for 16-bit gate and 32 bits for 32-bit gates *)
(* Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);
Push(TempEFlags);
Push(CS);
Push(EIP);
GS ← 0; (*segment registers nullified, invalid in protected mode *)
FS ← 0;
DS ← 0;
ES ← 0;
CS ← Gate(CS);
IF OperandSize=32
  THEN
    EIP ← Gate(instruction pointer);
  ELSE (* OperandSize is 16 *)
    EIP ← Gate(instruction pointer) AND 0000FFFFH;
  FI;
(* Starts execution of new routine in Protected Mode *)
END;
```

**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

INTRA-PRIVILEGE-LEVEL-INTERRUPT:

(\* PE=1, DPL ← CPL or conforming segment \*)

IF 32-bit gate

THEN

IF current stack does not have room for 16 bytes (error code pushed)

OR 12 bytes (no error code pushed); THEN #SS(0);

FI;

ELSE (\* 16-bit gate \*)

IF current stack does not have room for 8 bytes (error code pushed)

OR 6 bytes (no error code pushed); THEN #SS(0);

FI;

IF instruction pointer not within code segment limit THEN #GP(0); FI;

IF 32-bit gate

THEN

Push (EFLAGS);

Push (far pointer to return instruction); (\* 3 words padded to 4 \*)

CS:EIP ← Gate(CS:EIP); (\* segment descriptor information also loaded \*)

Push (ErrorCode); (\* if any \*)

ELSE (\* 16-bit gate \*)

Push (FLAGS);

Push (far pointer to return location); (\* 2 words \*)

CS:IP ← Gate(CS:IP); (\* segment descriptor information also loaded \*)

Push (ErrorCode); (\* if any \*)

FI;

CS(RPL) ← CPL;

IF interrupt gate

THEN

IF ← 0; FI;

TF ← 0;

NT ← 0;

VM ← 0;

RF ← 0;

FI;

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(0) If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.

**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

#GP(selector)	<p>If the segment selector in the interrupt-, trap-, or task gate is null.</p> <p>If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.</p> <p>If the interrupt vector number is outside the IDT limits.</p> <p>If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.</p> <p>If an interrupt is generated by the INT <i>n</i>, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.</p> <p>If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.</p>
#SS(selector)	<p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>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.</p>
#NP(selector)	<p>If code segment, interrupt-, trap-, or task gate, or TSS is not present.</p>
#TS(selector)	<p>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.</p> <p>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.</p> <p>If the stack segment selector in the TSS is null.</p> <p>If the stack segment for the TSS is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>

**Real-Address Mode Exceptions**

#GP	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the interrupt vector number is outside the IDT limits.</p>
-----	---

**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

#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.

**Virtual-8086 Mode Exceptions**

#GP(0) (For INT *n*, 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.

#GP(selector) 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 *n* 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(selector) 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(selector) If code segment, interrupt-, trap-, or task gate, or TSS is not present.

#TS(selector) 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.

**INT *n*/INTO/INT 3—Call to Interrupt Procedure (Continued)**

#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.

## INVD—Invalidate Internal Caches

Opcode	Instruction	Description
0F 08	INVD	Flush internal caches; initiate flushing of external caches.

### 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.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

### IA-32 Architecture Compatibility

The INVD instruction is implementation dependent, and its function may be implemented differently on future 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.

### Real-Address Mode Exceptions

None.

## **INVD—Invalidate Internal Caches (Continued)**

### **Virtual-8086 Mode Exceptions**

#GP(0)                    The INVD instruction cannot be executed in virtual-8086 mode.



## INVLPG—Invalidate TLB Entry

Opcode	Instruction	Description
OF 01/7	INVLPG <i>m</i>	Invalidate TLB Entry for page that contains <i>m</i>

### 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 of a program or procedure 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 flushes the entire TLB. See “MOV—Move to/from Control Registers” in this chapter for further information on operations that flush the TLB.

### IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on future 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.

### Real-Address Mode Exceptions

#UD Operand is a register.

### Virtual-8086 Mode Exceptions

#GP(0) The INVLPG instruction cannot be executed at the virtual-8086 mode.

## IRET/IRETD—Interrupt Return

Opcode	Instruction	Description
CF	IRET	Interrupt return (16-bit operand size)
CF	IRETD	Interrupt return (32-bit operand size)

### 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 6 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction performs a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack). As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

## IRET/IRETD—Interrupt Return (Continued)

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.

### Operation

```

IF PE ← 0
  THEN
    GOTO REAL-ADDRESS-MODE;;
  ELSE
    GOTO PROTECTED-MODE;
FI;

REAL-ADDRESS-MODE;
  IF OperandSize ← 32
    THEN
      IF top 12 bytes of stack not within stack limits THEN #SS; FI;
      IF instruction pointer not within code segment limits 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;
      IF instruction pointer not within code segment limits THEN #GP(0); FI;
      EIP ← Pop();
      EIP ← EIP AND 0000FFFFH;
      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

```

**IRET/IRETD—Interrupt Return (Continued)**

```

        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);
    FI;
    tempEIP ← Pop();
    tempCS ← Pop();
    tempEFLAGS ← Pop();
    tempEIP ← tempEIP AND FFFFH;
    tempEFLAGS ← tempEFLAGS AND FFFFH;
FI;
IF tempEFLAGS(VM) ← 1 AND CPL=0
    THEN
        GOTO RETURN-TO-VIRTUAL-8086-MODE;
        (* PE=1, VM=1 in EFLAGS image *)
    ELSE
        GOTO PROTECTED-MODE-RETURN;
        (* PE=1, VM=0 in EFLAGS image *)
FI;

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); FI;
            IF instruction pointer not within code segment limits THEN #GP(0); FI;
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            EFLAGS ← Pop();
            (*VM,IOPL,VIP,and VIF EFLAGS bits are not modified by pop *)
        ELSE (* OperandSize ← 16 *)
            IF top 6 bytes of stack are not within stack limits THEN #SS(0); FI;
            IF instruction pointer not within code segment limits THEN #GP(0); FI;
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)
            EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS is not modified by pop *)
        FI;
    ELSE
        #GP(0); (* trap to virtual-8086 monitor: PE=1, VM=1, IOPL<3 *)
FI;

```

**IRET/IRETD—Interrupt Return (Continued)**

END;

RETURN-TO-VIRTUAL-8086-MODE:

(\* Interrupted procedure was in virtual-8086 mode: PE=1, VM=1 in flags image \*)

IF top 24 bytes of stack are not within stack segment limits  
THEN #SS(0);

FI;

IF instruction pointer not within code segment limits  
THEN #GP(0);

FI;

CS ← tempCS;

EIP ← tempEIP;

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;

(\* Resume execution in Virtual-8086 mode \*)

END;

TASK-RETURN: (\* PE=1, VM=1, 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 #GP(TSS selector);

FI;

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 #GP(TSS selector);

FI;

IF TSS not present  
THEN #NP(TSS selector);

FI;

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);

FI;

END;

PROTECTED-MODE-RETURN: (\* PE=1, VM=0 in flags image \*)

IF return code segment selector is null THEN GP(0); FI;

IF return code segment selector addresses descriptor beyond descriptor table limit

**IRET/IRETD—Interrupt Return (Continued)**

```

    THEN GP(selector); FI;
    Read segment descriptor pointed to by the return code segment selector
    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 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;
END;
```

```

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE=1, VM=0 in flags image, RPL=CPL *)
    IF EIP is not within code segment limits THEN #GP(0); FI;
    EIP ← tempEIP;
    CS ← tempCS; (* segment descriptor information also loaded *)
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize=32
        THEN
            EFLAGS(RF, AC, ID) ← tempEFLAGS;
    FI;
    IF CPL ≤ IOPL
        THEN
            EFLAGS(IF) ← tempEFLAGS;
    FI;
    IF CPL ← 0
        THEN
            EFLAGS(IOPL) ← tempEFLAGS;
            IF OperandSize=32
                THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS;
    FI;
    FI;
END;
```

```

RETURN-TO-OUTER-PRIVILGE-LEVEL:
    IF OperandSize=32
        THEN
            IF top 8 bytes on stack are not within limits THEN #SS(0); FI;
            ELSE (* OperandSize=16 *)
                IF top 4 bytes on stack are not within limits THEN #SS(0); FI;
    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
```

**IRET/IRETD—Interrupt Return (Continued)**

```

    THEN #GP(SSselector); FI;
  Read segment descriptor pointed to by return segment selector;
  IF stack segment selector RPL ≠ RPL of the return code segment selector
    IF stack segment selector RPL ≠ RPL of the return code segment selector
      OR the stack segment descriptor does not indicate a writable data segment;
      OR stack segment DPL ≠ RPL of the return code segment selector
        THEN #GP(SS selector);
    FI;
    IF stack segment is not present THEN #SS(SS selector); FI;
  IF tempEIP is not within code segment limit THEN #GP(0); FI;
  EIP ← tempEIP;
  CS ← tempCS;
  EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
  IF OperandSize=32
    THEN
      EFLAGS(RF, AC, ID) ← tempEFLAGS;
    FI;
  IF CPL ≤ IOPL
    THEN
      EFLAGS(IF) ← tempEFLAGS;
    FI;
  IF CPL ← 0
    THEN
      EFLAGS(IOPL) ← tempEFLAGS;
      IF OperandSize=32
        THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS;
      FI;
    FI;
  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 *)
        FI;
    OD;
  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.

**IRET/IRETD—Interrupt Return (Continued)****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 greater 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 busy or 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.

**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.



**IRET/IRETD—Interrupt Return (Continued)**

- #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.

## Jcc—Jump if Condition Is Met

Opcode	Instruction	Description
77 <i>cb</i>	JA <i>rel8</i>	Jump short if above (CF=0 and ZF=0)
73 <i>cb</i>	JAE <i>rel8</i>	Jump short if above or equal (CF=0)
72 <i>cb</i>	JB <i>rel8</i>	Jump short if below (CF=1)
76 <i>cb</i>	JBE <i>rel8</i>	Jump short if below or equal (CF=1 or ZF=1)
72 <i>cb</i>	JC <i>rel8</i>	Jump short if carry (CF=1)
E3 <i>cb</i>	JCXZ <i>rel8</i>	Jump short if CX register is 0
E3 <i>cb</i>	JECXZ <i>rel8</i>	Jump short if ECX register is 0
74 <i>cb</i>	JE <i>rel8</i>	Jump short if equal (ZF=1)
7F <i>cb</i>	JG <i>rel8</i>	Jump short if greater (ZF=0 and SF=OF)
7D <i>cb</i>	JGE <i>rel8</i>	Jump short if greater or equal (SF=OF)
7C <i>cb</i>	JL <i>rel8</i>	Jump short if less (SF<>OF)
7E <i>cb</i>	JLE <i>rel8</i>	Jump short if less or equal (ZF=1 or SF<>OF)
76 <i>cb</i>	JNA <i>rel8</i>	Jump short if not above (CF=1 or ZF=1)
72 <i>cb</i>	JNAE <i>rel8</i>	Jump short if not above or equal (CF=1)
73 <i>cb</i>	JNB <i>rel8</i>	Jump short if not below (CF=0)
77 <i>cb</i>	JNBE <i>rel8</i>	Jump short if not below or equal (CF=0 and ZF=0)
73 <i>cb</i>	JNC <i>rel8</i>	Jump short if not carry (CF=0)
75 <i>cb</i>	JNE <i>rel8</i>	Jump short if not equal (ZF=0)
7E <i>cb</i>	JNG <i>rel8</i>	Jump short if not greater (ZF=1 or SF<>OF)
7C <i>cb</i>	JNGE <i>rel8</i>	Jump short if not greater or equal (SF<>OF)
7D <i>cb</i>	JNL <i>rel8</i>	Jump short if not less (SF=OF)
7F <i>cb</i>	JNLE <i>rel8</i>	Jump short if not less or equal (ZF=0 and SF=OF)
71 <i>cb</i>	JNO <i>rel8</i>	Jump short if not overflow (OF=0)
7B <i>cb</i>	JNP <i>rel8</i>	Jump short if not parity (PF=0)
79 <i>cb</i>	JNS <i>rel8</i>	Jump short if not sign (SF=0)
75 <i>cb</i>	JNZ <i>rel8</i>	Jump short if not zero (ZF=0)
70 <i>cb</i>	JO <i>rel8</i>	Jump short if overflow (OF=1)
7A <i>cb</i>	JP <i>rel8</i>	Jump short if parity (PF=1)
7A <i>cb</i>	JPE <i>rel8</i>	Jump short if parity even (PF=1)
7B <i>cb</i>	JPO <i>rel8</i>	Jump short if parity odd (PF=0)
78 <i>cb</i>	JS <i>rel8</i>	Jump short if sign (SF=1)
74 <i>cb</i>	JZ <i>rel8</i>	Jump short if zero (ZF ← 1)
0F 87 <i>cw/cd</i>	JA <i>rel16/32</i>	Jump near if above (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	JAE <i>rel16/32</i>	Jump near if above or equal (CF=0)
0F 82 <i>cw/cd</i>	JB <i>rel16/32</i>	Jump near if below (CF=1)
0F 86 <i>cw/cd</i>	JBE <i>rel16/32</i>	Jump near if below or equal (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	JC <i>rel16/32</i>	Jump near if carry (CF=1)
0F 84 <i>cw/cd</i>	JE <i>rel16/32</i>	Jump near if equal (ZF=1)
0F 84 <i>cw/cd</i>	JZ <i>rel16/32</i>	Jump near if 0 (ZF=1)
0F 8F <i>cw/cd</i>	JG <i>rel16/32</i>	Jump near if greater (ZF=0 and SF=OF)

## Jcc—Jump if Condition Is Met (Continued)

Opcode	Instruction	Description
0F 8D <i>cw/cd</i>	JGE <i>rel16/32</i>	Jump near if greater or equal (SF=OF)
0F 8C <i>cw/cd</i>	JL <i>rel16/32</i>	Jump near if less (SF<>OF)
0F 8E <i>cw/cd</i>	JLE <i>rel16/32</i>	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 <i>cw/cd</i>	JNA <i>rel16/32</i>	Jump near if not above (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	JNAE <i>rel16/32</i>	Jump near if not above or equal (CF=1)
0F 83 <i>cw/cd</i>	JNB <i>rel16/32</i>	Jump near if not below (CF=0)
0F 87 <i>cw/cd</i>	JNBE <i>rel16/32</i>	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	JNC <i>rel16/32</i>	Jump near if not carry (CF=0)
0F 85 <i>cw/cd</i>	JNE <i>rel16/32</i>	Jump near if not equal (ZF=0)
0F 8E <i>cw/cd</i>	JNG <i>rel16/32</i>	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C <i>cw/cd</i>	JNGE <i>rel16/32</i>	Jump near if not greater or equal (SF<>OF)
0F 8D <i>cw/cd</i>	JNL <i>rel16/32</i>	Jump near if not less (SF=OF)
0F 8F <i>cw/cd</i>	JNLE <i>rel16/32</i>	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 <i>cw/cd</i>	JNO <i>rel16/32</i>	Jump near if not overflow (OF=0)
0F 8B <i>cw/cd</i>	JNP <i>rel16/32</i>	Jump near if not parity (PF=0)
0F 89 <i>cw/cd</i>	JNS <i>rel16/32</i>	Jump near if not sign (SF=0)
0F 85 <i>cw/cd</i>	JNZ <i>rel16/32</i>	Jump near if not zero (ZF=0)
0F 80 <i>cw/cd</i>	JO <i>rel16/32</i>	Jump near if overflow (OF=1)
0F 8A <i>cw/cd</i>	JP <i>rel16/32</i>	Jump near if parity (PF=1)
0F 8A <i>cw/cd</i>	JPE <i>rel16/32</i>	Jump near if parity even (PF=1)
0F 8B <i>cw/cd</i>	JPO <i>rel16/32</i>	Jump near if parity odd (PF=0)
0F 88 <i>cw/cd</i>	JS <i>rel16/32</i>	Jump near if sign (SF=1)
0F 84 <i>cw/cd</i>	JZ <i>rel16/32</i>	Jump near if 0 (ZF=1)

### 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 to 0s, resulting in a maximum instruction pointer size of 16 bits.

## Jcc—Jump if Condition Is Met (Continued)

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 *JECXZ* and *JCXZ* instructions differs from the other *Jcc* instructions because they do not check the status flags. Instead they check the contents of the *ECX* and *CX* registers, respectively, for 0. Either the *CX* or *ECX* register is chosen according to the address-size attribute. These instructions are useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as *LOOPNE*). They prevent entering the loop when the *ECX* or *CX* register is equal to 0, which would cause the loop to execute  $2^{32}$  or 64K times, respectively, instead of zero times.

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

### Operation

```
IF condition
  THEN
    EIP ← EIP + SignExtend(DEST);
    IF OperandSize ← 16
      THEN
        EIP ← EIP AND 0000FFFFH;
      FI;
    ELSE (* OperandSize = 32 *)
      IF EIP < CS.Base OR EIP > CS.Limit
        #GP
      FI;
  FI;
```

## Jcc—Jump if Condition Is Met (Continued)

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the offset being jumped to is beyond the limits of the CS segment.

### 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.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

## JMP—Jump

Opcode	Instruction	Description
EB <i>cb</i>	JMP <i>rel8</i>	Jump short, relative, displacement relative to next instruction
E9 <i>cw</i>	JMP <i>rel16</i>	Jump near, relative, displacement relative to next instruction
E9 <i>cd</i>	JMP <i>rel32</i>	Jump near, relative, displacement relative to next instruction
FF /4	JMP <i>r/m16</i>	Jump near, absolute indirect, address given in <i>r/m16</i>
FF /4	JMP <i>r/m32</i>	Jump near, absolute indirect, address given in <i>r/m32</i>
EA <i>cd</i>	JMP <i>ptr16:16</i>	Jump far, absolute, address given in operand
EA <i>cp</i>	JMP <i>ptr16:32</i>	Jump far, absolute, address given in operand
FF /5	JMP <i>m16:16</i>	Jump far, absolute indirect, address given in <i>m16:16</i>
FF /5	JMP <i>m16:32</i>	Jump far, absolute indirect, address given in <i>m16:32</i>

### 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 6, *Task Management*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, 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 to 0s, resulting in a maximum instruction pointer size of 16 bits.

## JMP—Jump (Continued)

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 to 0s.

**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.

## JMP—Jump (Continued)

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 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 6, *Task Management*, in *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, 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.

### Operation

IF near jump

    THEN IF near relative jump

        THEN

            tempEIP ← EIP + DEST; (\* EIP is instruction following JMP instruction\*)

        ELSE (\* near absolute jump \*)

            tempEIP ← DEST;

    FI;

    IF tempEIP is beyond code segment limit THEN #GP(0); FI;

    IF OperandSize ← 32

        THEN

            EIP ← tempEIP;

        ELSE (\* OperandSize=16 \*)

            EIP ← tempEIP AND 0000FFFFH;

    FI;

FI:

IF far jump AND (PE ← 0 OR (PE ← 1 AND VM ← 1)) (\* real-address or virtual-8086 mode \*)



**JMP—Jump (Continued)**

THEN

tempEIP ← DEST[offset]; (\* DEST is *ptr16:32* or [*m16:32*] \*)

IF tempEIP is beyond code segment limit THEN #GP(0); FI;

CS ← DEST[segment selector]; (\* DEST is *ptr16:32* or [*m16:32*] \*)

IF OperandSize ← 32

THEN

EIP ← tempEIP; (\* DEST is *ptr16:32* or [*m16:32*] \*)

ELSE (\* OperandSize ← 16 \*)

EIP ← tempEIP AND 0000FFFFH; (\* clear upper 16 bits \*)

FI;

FI;

IF far jump AND (PE ← 1 AND VM ← 0) (\* 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 segment type is not a conforming or nonconforming code segment, call gate,  
task gate, or TSS THEN #GP(segment selector); 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 DPL &gt; 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 tempEIP not in code segment limit THEN #GP(0); FI;

CS ← DEST[SegmentSelector]; (\* segment descriptor information also loaded \*)

CS(RPL) ← CPL

EIP ← tempEIP;

END;

**JMP—Jump (Continued)**

## NONCONFORMING-CODE-SEGMENT:

```

IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(code segment selector); FI;
IF segment not present THEN #NP(segment selector); FI;
IF instruction pointer outside code segment limit THEN #GP(0); FI;
tempEIP ← DEST[offset];
IF OperandSize=16
    THEN tempEIP ← tempEIP AND 0000FFFFH;
FI;
IF tempEIP not in code segment limit THEN #GP(0); FI;
CS ← DEST[SegmentSelector]; (* 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); FI;
IF call gate not present THEN #NP(call gate selector); FI;
IF call gate code-segment selector is null THEN #GP(0); FI;
IF call gate code-segment selector index is outside descriptor table limits
    THEN #GP(code segment selector); FI;
Read code segment descriptor;
IF code-segment segment descriptor does not indicate a code segment
    OR code-segment segment descriptor is conforming and DPL > CPL
    OR code-segment segment descriptor is non-conforming and DPL ≠ CPL
    THEN #GP(code segment selector); FI;
IF code segment is not present THEN #NP(code-segment selector); FI;
IF instruction pointer is not within code-segment limit THEN #GP(0); FI;
tempEIP ← DEST[offset];
IF GateSize=16
    THEN tempEIP ← tempEIP AND 0000FFFFH;
FI;
IF tempEIP not in code segment limit THEN #GP(0); FI;
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); FI;
IF task gate not present THEN #NP(gate selector); FI;
Read the TSS segment selector in the task-gate descriptor;

```

**JMP—Jump (Continued)**

```

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); FI;
IF TSS not present THEN #NP(TSS selector); FI;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit THEN #GP(0); FI;
END;
```

**TASK-STATE-SEGMENT:**

```

IF TSS DPL < CPL
  OR TSS DPL < TSS segment-selector RPL
  OR TSS descriptor indicates TSS not available
    THEN #GP(TSS selector); FI;
IF TSS is not present THEN #NP(TSS selector); FI;
SWITCH-TASKS to TSS
IF EIP not within code segment limit THEN #GP(0); FI;
END;
```

**Flags Affected**

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

**Protected Mode Exceptions**

#GP(0)	<p>If offset in target operand, call gate, or TSS is beyond the code segment limits.</p> <p>If the segment selector in the destination operand, call gate, task gate, or TSS is null.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>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.</p> <p>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.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p>

**JMP—Jump (Continued)**

	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 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.)

**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.

**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.)

## LAHF—Load Status Flags into AH Register

Opcode	Instruction	Description
9F	LAHF	Load: AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF)

### Description

Moves the low byte of the EFLAGS register (which includes status flags SF, ZF, AF, PF, and CF) to the AH register. Reserved bits 1, 3, and 5 of the EFLAGS register are set in the AH register as shown in the “Operation” section below.

### Operation

AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);

### Flags Affected

None (that is, the state of the flags in the EFLAGS register is not affected).

### Exceptions (All Operating Modes)

None.

## LAR—Load Access Rights Byte

Opcode	Instruction	Description
0F 02 /r	LAR <i>r16,r/m16</i>	<i>r16</i> ← <i>r/m16</i> masked by FF00H
0F 02 /r	LAR <i>r32,r/m32</i>	<i>r32</i> ← <i>r/m32</i> masked by 00FxFF00H

### Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. 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.

When the operand size is 32 bits, the access rights for a segment descriptor include the type and DPL fields and the S, P, AVL, D/B, and G flags, all of which are located in the second doubleword (bytes 4 through 7) of the segment descriptor. The doubleword is masked by 00FxFF00H before it is loaded into the destination operand. When the operand size is 16 bits, the access rights include the type and DPL fields. Here, the two lower-order bytes of the doubleword are masked by FF00H before being loaded into the destination operand.

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 the following table.
- 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.

## LAR—Load Access Rights Byte (Continued)

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	Yes
5	16-bit/32-bit task gate	Yes
6	16-bit interrupt gate	No
7	16-bit trap gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	Yes
D	Reserved	No
E	32-bit interrupt gate	No
F	32-bit trap gate	No

### Operation

```

IF SRC[Offset] > descriptor table limit THEN ZF ← 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
    AND (CPL > DPL) OR (RPL > DPL)
    OR Segment type is not valid for instruction
    THEN
        ZF ← 0
    ELSE
        IF OperandSize ← 32
            THEN
                DEST ← [SRC] AND 00FF00H;
            ELSE (*OperandSize ← 16*)
                DEST ← [SRC] AND FF00H;
        FI;
    FI;

```

### Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

## LAR—Load Access Rights Byte (Continued)

### 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. (Only occurs when fetching target from memory.)

### 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.
-----	--



## LDMXCSR—Load MXCSR Register

Opcode	Instruction	Description
0F,AE,/2	LDMXCSR <i>m32</i>	Load MXCSR register from <i>m32</i> .

### Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See “MXCSR Control and Status Register” in Chapter 10, of the *IA-32 Intel Architecture 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 STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a 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 SSE or SSE2 instruction that causes that particular SIMD floating-point exception to be reported.

### Operation

$\text{MXCSR} \leftarrow m32;$

### C/C++ Compiler Intrinsic Equivalent

`_mm_setcsr(unsigned int i)`

### Numeric 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.
#NM	If TS in CR0 is set.

**LDMXCSR—Load MXCSR Register (Continued)**

#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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**

Interrupt 13	If any part of the operand would lie outside of the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

**Virtual 8086 Mode Exceptions**

Same exceptions as in Real Address Mode.

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## LDS/LES/LFS/LGS/LSS—Load Far Pointer

<b>Opcode</b>	<b>Instruction</b>	<b>Description</b>
<i>C5 /r</i>	<i>LDS r16,m16:16</i>	Load DS: <i>r16</i> with far pointer from memory
<i>C5 /r</i>	<i>LDS r32,m16:32</i>	Load DS: <i>r32</i> with far pointer from memory
<i>0F B2 /r</i>	<i>LSS r16,m16:16</i>	Load SS: <i>r16</i> with far pointer from memory
<i>0F B2 /r</i>	<i>LSS r32,m16:32</i>	Load SS: <i>r32</i> with far pointer from memory
<i>C4 /r</i>	<i>LES r16,m16:16</i>	Load ES: <i>r16</i> with far pointer from memory
<i>C4 /r</i>	<i>LES r32,m16:32</i>	Load ES: <i>r32</i> with far pointer from memory
<i>0F B4 /r</i>	<i>LFS r16,m16:16</i>	Load FS: <i>r16</i> with far pointer from memory
<i>0F B4 /r</i>	<i>LFS r32,m16:32</i>	Load FS: <i>r32</i> with far pointer from memory
<i>0F B5 /r</i>	<i>LGS r16,m16:16</i>	Load GS: <i>r16</i> with far pointer from memory
<i>0F B5 /r</i>	<i>LGS r32,m16:32</i>	Load GS: <i>r32</i> with far pointer from memory

### 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.)

### Operation

```

IF ProtectedMode
  THEN IF SS is loaded
    THEN IF SegmentSelector ← null
      THEN #GP(0);
    FI;
  ELSE IF Segment selector index is not within descriptor table limits
    OR Segment selector RPL ≠ CPL
    OR Access rights indicate nonwritable data segment
    OR DPL ≠ CPL

```

**LDS/LES/LFS/LGS/LSS—Load Far Pointer (Continued)**

```

    THEN #GP(selector);
  FI;
  ELSE IF Segment marked not present
    THEN #SS(selector);
  FI;
  SS ← SegmentSelector(SRC);
  SS ← SegmentDescriptor([SRC]);
  ELSE IF DS, ES, FS, or GS is loaded with non-null segment selector
    THEN IF Segment selector index is not within descriptor table limits
      OR Access rights indicate segment neither data nor readable code segment
      OR (Segment is data or nonconforming-code segment
          AND both RPL and CPL > DPL)
        THEN #GP(selector);
  FI;
  ELSE IF Segment marked not present
    THEN #NP(selector);
  FI;
  SegmentRegister ← SegmentSelector(SRC) AND RPL;
  SegmentRegister ← SegmentDescriptor([SRC]);
  ELSE IF DS, ES, FS, or GS is loaded with a null selector:
    SegmentRegister ← NullSelector;
    SegmentRegister(DescriptorValidBit) ← 0; (*hidden flag; not accessible by software*)
  FI;
FI;
IF (Real-Address or Virtual-8086 Mode)
  THEN
    SegmentRegister ← SegmentSelector(SRC);
FI;
DEST ← Offset(SRC);

```

**Flags Affected**

None.

**Protected Mode Exceptions**

#UD	If source operand is not a memory location.
#GP(0)	If a null selector is loaded into the SS register.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

**LDS/LES/LFS/LGS/LSS—Load Far Pointer (Continued)**

#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 nonwritable data segment, or DPL is not equal to CPL.  If the DS, ES, FS, or GS register is being loaded with a non-null segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or 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-null segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If source operand is not a memory location.

**Virtual-8086 Mode Exceptions**

#UD	If source 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.

## LEA—Load Effective Address

Opcode	Instruction	Description
8D /r	LEA <i>r16,m</i>	Store effective address for <i>m</i> in register <i>r16</i>
8D /r	LEA <i>r32,m</i>	Store effective address for <i>m</i> in register <i>r32</i>

### Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

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.

### 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 *)
  ELSE IF OperandSize ← 32 AND AddressSize ← 16
    THEN
      temp ← EffectiveAddress(SRC); (* 16-bit address *)
      DEST ← ZeroExtend(temp); (* 32-bit address *)
  ELSE IF OperandSize ← 32 AND AddressSize ← 32
    THEN

```

**LEA—Load Effective Address (Continued)**

DEST ← EffectiveAddress(SRC); (\* 32-bit address \*)  
FI;  
FI;

**Flags Affected**

None.

**Protected Mode Exceptions**

#UD If source operand is not a memory location.

**Real-Address Mode Exceptions**

#UD If source operand is not a memory location.

**Virtual-8086 Mode Exceptions**

#UD If source operand is not a memory location.

## LEAVE—High Level Procedure Exit

Opcode	Instruction	Description
C9	LEAVE	Set SP to BP, then pop BP
C9	LEAVE	Set ESP to EBP, then pop EBP

### Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See “Procedure Calls for Block-Structured Languages” in Chapter 6 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for detailed information on the use of the ENTER and LEAVE instructions.

### Operation

```
IF StackAddressSize ← 32
  THEN
    ESP ← EBP;
  ELSE (* StackAddressSize ← 16*)
    SP ← BP;
```

FI;

```
IF OperandSize ← 32
  THEN
    EBP ← Pop();
  ELSE (* OperandSize ← 16*)
    BP ← Pop();
```

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.



## LEAVE—High Level Procedure Exit (Continued)

#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 the EBP register points to a location outside of the effective address space from 0 to FFFFH.

### 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.

## **LES—Load Full Pointer**

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

## LFENCE—Load Fence

Opcode	Instruction	Description
OF AE /5	LFENCE	Serializes load operations.

### Description

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. This serializing operation guarantees that every load instruction that precedes in program order the LFENCE instruction is globally visible before any load instruction that follows the LFENCE instruction is globally visible. The LFENCE instruction is ordered with respect to load instructions, other LFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to store instructions or the SFENCE instruction.

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 insuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

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). The PREFETCH $h$  instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the LFENCE instruction is not ordered with respect to PREFETCH $h$  instructions or any other speculative fetching mechanism (that is, data could be speculative loaded into the cache just before, during, or after the execution of an LFENCE instruction).

### Operation

```
Wait_On_Following_Loads_Until(preceding_loads_globally_visible);
```

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

```
void_mm_lfence(void)
```

### Exceptions (All Modes of Operation)

None.

## **LFS—Load Full Pointer**

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

## LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Description
OF 01 /2	LGDT <i>m16&amp;32</i>	Load <i>m</i> into GDTR
OF 01 /3	LIDT <i>m16&amp;32</i>	Load <i>m</i> into IDTR

### 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.

See “SFENCE—Store Fence” in this chapter 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 00FFFFFFFH;
      ELSE (* 32-bit Operand Size *)
        IDTR(Limit) ← SRC[0:15];
        IDTR(Base) ← SRC[16:47];
    FI;
  ELSE (* instruction is LGDT *)
    IF OperandSize ← 16
      THEN
        GDTR(Limit) ← SRC[0:15];
        GDTR(Base) ← SRC[16:47] AND 00FFFFFFFH;
      ELSE (* 32-bit Operand Size *)
        GDTR(Limit) ← SRC[0:15];
        GDTR(Base) ← SRC[16:47];
    FI; FI;

```

## LGDT/LIDT—Load Global/Interrupt Descriptor Table Register (Continued)

### Flags Affected

None.

### Protected Mode Exceptions

#UD	If source operand is not a memory location.
#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.
#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.
--------	---

## **LGS—Load Full Pointer**

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

## LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Description
0F 00 /2	LLDT <i>r/m16</i>	Load segment selector <i>r/m16</i> into LDTR

### 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 the source operand is 0, the LDTR is marked invalid and all 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. Also, this instruction can only be executed in protected mode.

### Operation

```
IF SRC[Offset] > descriptor table limit THEN #GP(segment selector); FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ LDT THEN #GP(segment selector); FI;
IF segment descriptor is not present THEN #NP(segment selector);
LDTR(SegmentSelector) ← SRC;
LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;
```

### Flags Affected

None.

### Protected Mode Exceptions

- |               |   |
|---------------|---|
| #GP(0)        | <p>If the current privilege level is not 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains a null segment selector.</p> |
| #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.   |



**LLDT—Load Local Descriptor Table Register (Continued)**

	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.

**Real-Address Mode Exceptions**

#UD	The LLDT instruction is not recognized in real-address mode.
-----	--

**Virtual-8086 Mode Exceptions**

#UD	The LLDT instruction is recognized in virtual-8086 mode.
-----	--

## **LIDT—Load Interrupt Descriptor Table Register**

See entry for LGDT/LIDT—Load Global/Interrupt Descriptor Table Register.

## LMSW—Load Machine Status Word

Opcode	Instruction	Description
OF 01 /6	LMSW <i>r/m16</i>	Loads <i>r/m16</i> in machine status word of CR0

### Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used 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, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

### Operation

CR0[0:3] ← SRC[0:3];

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

## LMSW—Load Machine Status Word (Continued)

### Real-Address Mode Exceptions

#GP                    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

### Virtual-8086 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.

#SS(0)                If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)      If a page fault occurs.

## LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Description
F0	LOCK	Asserts LOCK# signal for duration of the accompanying instruction

### 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 insures that the processor has exclusive use of any shared memory while the signal is asserted.

Note that in later IA-32 processors (such as the Pentium Pro processor), locking may occur without the LOCK# signal being asserted. See IA-32 Architecture Compatibility 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, 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 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.

### IA-32 Architecture Compatibility

Beginning with the Pentium Pro processor, 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 insures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 7 of *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information on locking of caches.

### Operation

AssertLOCK#/(DurationOfAccompanyingInstruction)

### Flags Affected

None.

## LOCK—Assert LOCK# Signal Prefix (Continued)

### Protected Mode Exceptions

#UD                    If the LOCK prefix is used with an instruction not listed in the “Description” section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

### Real-Address Mode Exceptions

#UD                    If the LOCK prefix is used with an instruction not listed in the “Description” section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

### Virtual-8086 Mode Exceptions

#UD                    If the LOCK prefix is used with an instruction not listed in the “Description” section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

## LODS/LODSB/LODSW/LODSD—Load String

Opcode	Instruction	Description
AC	LODS m8	Load byte at address DS:(E)SI into AL
AD	LODS m16	Load word at address DS:(E)SI into AX
AD	LODS m32	Load doubleword at address DS:(E)SI into EAX
AC	LODSB	Load byte at address DS:(E)SI into AL
AD	LODSW	Load word at address DS:(E)SI into AX
AD	LODSD	Load doubleword at address DS:(E)SI into EAX

### 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:EDI 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.

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 this chapter for a description of the REP prefix.

**LODS/LODSB/LODSW/LODSD—Load String (Continued)****Operation**

```

IF (byte load)
  THEN
    AL ← SRC; (* byte load *)
    THEN IF DF ← 0
      THEN (E)SI ← (E)SI + 1;
      ELSE (E)SI ← (E)SI - 1;
    FI;
  ELSE IF (word load)
    THEN
      AX ← SRC; (* word load *)
      THEN IF DF ← 0
        THEN (E)SI ← (E)SI + 2;
        ELSE (E)SI ← (E)SI - 2;
      FI;
    ELSE (* doubleword transfer *)
      EAX ← SRC; (* doubleword load *)
      THEN 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 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**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
-----	---



**LODS/LODSB/LODSW/LODSD—Load String (Continued)**

#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.

## LOOP/LOOP $cc$ —Loop According to ECX Counter

Opcode	Instruction	Description
E2 <i>cb</i>	LOOP <i>rel8</i>	Decrement count; jump short if count $\neq$ 0
E1 <i>cb</i>	LOOPE <i>rel8</i>	Decrement count; jump short if count $\neq$ 0 and ZF=1
E1 <i>cb</i>	LOOPZ <i>rel8</i>	Decrement count; jump short if count $\neq$ 0 and ZF=1
E0 <i>cb</i>	LOOPNE <i>rel8</i>	Decrement count; jump short if count $\neq$ 0 and ZF=0
E0 <i>cb</i>	LOOPNZ <i>rel8</i>	Decrement count; jump short if count $\neq$ 0 and ZF=0

### Description

Performs a loop operation using the ECX or CX register as a counter. 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. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

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). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of  $-128$  to  $+127$  are allowed with this instruction.

Some forms of the loop instruction (LOOP $cc$ ) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code ( $cc$ ) is associated with each instruction to indicate the condition being tested for. Here, the LOOP $cc$  instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

### Operation

```
IF AddressSize  $\leftarrow$  32
  THEN
    Count is ECX;
  ELSE (* AddressSize  $\leftarrow$  16 *)
    Count is CX;
FI;
Count  $\leftarrow$  Count - 1;
```

```
IF instruction is not LOOP
  THEN
    IF (instruction  $\leftarrow$  LOOPE) OR (instruction  $\leftarrow$  LOOPZ)
      THEN
        IF (ZF = 1) AND (Count  $\neq$  0)
          THEN BranchCond  $\leftarrow$  1;
          ELSE BranchCond  $\leftarrow$  0;
```

**LOOP/LOOPcc—Loop According to ECX Counter (Continued)**

```

    FI;
  FI;
  IF (instruction ← LOOPNE) OR (instruction ← LOOPNZ)
    THEN
      IF (ZF = 0) AND (Count ≠ 0)
        THEN BranchCond ← 1;
        ELSE BranchCond ← 0;
      FI;
    FI;
  ELSE (* instruction ← LOOP *)
    IF (Count ≠ 0)
      THEN BranchCond ← 1;
      ELSE BranchCond ← 0;
    FI;
  FI;
  IF BranchCond ← 1
    THEN
      EIP ← EIP + SignExtend(DEST);
      IF OperandSize ← 16
        THEN
          EIP ← EIP AND 0000FFFFH;
        ELSE (* OperandSize = 32 *)
          IF EIP < CS.Base OR EIP > CS.Limit
            #GP
          FI;
        ELSE
          Terminate loop and continue program execution at EIP;
        FI;
  FI;

```

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0) If the offset being jumped to is beyond the limits of the CS segment.

**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.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

## LSL—Load Segment Limit

Opcode	Instruction	Description
0F 03 /r	LSL <i>r16,r/m16</i>	Load: <i>r16</i> ← segment limit, selector <i>r/m16</i>
0F 03 /r	LSL <i>r32,r/m32</i>	Load: <i>r32</i> ← segment limit, selector <i>r/m32</i> )

### 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.

## LSL—Load Segment Limit (Continued)

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	No
5	16-bit/32-bit task gate	No
6	16-bit interrupt gate	No
7	16-bit trap gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	No
D	Reserved	No
E	32-bit interrupt gate	No
F	32-bit trap gate	No

### Operation

```

IF SRC[Offset] > descriptor table limit
    THEN ZF ← 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
    AND (CPL > DPL) OR (RPL > DPL)
    OR Segment type is not valid for instruction
    THEN
        ZF ← 0
    ELSE
        temp ← SegmentLimit([SRC]);
        IF (G ← 1)
            THEN
                temp ← ShiftLeft(12, temp) OR 00000FFFH;
        FI;
        IF OperandSize ← 32
            THEN
                DEST ← temp;
            ELSE (*OperandSize ← 16*)
    
```

## LSL—Load Segment Limit (Continued)

DEST ← temp AND FFFFH;  
FI;  
FI;

### Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is 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 an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

- #UD The LSL instruction is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

- #UD The LSL instruction is not recognized in virtual-8086 mode.

## **LSS—Load Full Pointer**

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

## LTR—Load Task Register

Opcode	Instruction	Description
0F 00 /3	LTR <i>r/m16</i>	Load <i>r/m16</i> into task register

### 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.

### Operation

IF SRC[Offset] > descriptor table limit OR IF SRC[type] ≠ global  
THEN #GP(segment selector);

FI;

Read segment descriptor;

IF segment descriptor is not for an available TSS THEN #GP(segment selector); FI;

IF segment descriptor is not present THEN #NP(segment selector);

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 DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.



## LTR—Load Task Register (Continued)

#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.

### 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.
-----	---

## MASKMOVDQU—Store Selected Bytes of Double Quadword

Opcode	Instruction	Description
66 0F F7 /r	MASKMOVDQU <i>xmm1</i> , <i>xmm2</i>	Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> .

### Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The location of the first byte of the memory location is specified by DI/EDI and DS registers. The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVEDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVEDQU 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.

## MASKMOVDQU—Store Selected Bytes of Double Quadword (Continued)

### Operation

```

IF (MASK[7] = 1)
  THEN DEST[DI/EDI] ← SRC[7-0] ELSE * memory location unchanged *; FI;
IF (MASK[15] = 1)
  THEN DEST[DI/EDI+1] ← SRC[15-8] ELSE * memory location unchanged *; FI;
  * Repeat operation for 3rd through 14th bytes in source operand *;
IF (MASK[127] = 1)
  THEN DEST[DI/EDI+15] ← SRC[127-120] ELSE * memory location unchanged *; FI;

```

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

```
void_mm_maskmoveu_si128(__m128i d, __m128i n, char * p)
```

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. (even if mask is all 0s).
#SS(0)	For an illegal address in the SS segment (even if mask is all 0s).
#PF(fault-code)	For a page fault (implementation specific).
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH. (even if mask is all 0s).
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault (implementation specific).
-----------------	---

## MASKMOVQ—Store Selected Bytes of Quadword

Opcode	Instruction	Description
0F F7 /r	MASKMOVQ <i>mm1</i> , <i>mm2</i>	Selectively write bytes from <i>mm1</i> to memory location using the byte mask in <i>mm2</i>

### 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 registers. The location of the first byte of the memory location is specified by DI/EDI and DS registers. (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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVEDQU 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 state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX 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 (implementations 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.

## MASKMOVQ—Store Selected Bytes of Quadword (Continued)

### Operation

```

IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7-0] ELSE * memory location unchanged *; FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI+1] ← SRC[15-8] ELSE * memory location unchanged *; FI;
    * Repeat operation for 3rd through 6th bytes in source operand *;
IF (MASK[63] = 1)
    THEN DEST[DI/EDI+15] ← SRC[63-56] ELSE * memory location unchanged *; FI;

```

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

```
void_mm_maskmove_si64(__m64d, __m64n, char * p)
```

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. (even if mask is all 0s).
#SS(0)	For an illegal address in the SS segment (even if mask is all 0s).
#PF(fault-code)	For a page fault (implementation specific).
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0. If Mod field of the ModR/M byte not 11B
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH. (even if mask is all 0s).
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

**MASKMOVQ—Store Selected Bytes of Quadword (Continued)****Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault (implementation specific).

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## MAXPD—Return Maximum Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 5F /r	MAXPD <i>xmm1</i> , <i>xmm2/m128</i>	Return the maximum double-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a SIMD compare of the packed double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of values to the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

If the values being compared are both 0.0s, 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. This behavior allows compilers to use the MAXPD instruction for common C conditional constructs. 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 the MAXPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

### Operation

```

DEST[63-0] ← IF ((DEST[63-0] == 0.0) AND (SRC[63-0] == 0.0)) THEN SRC[63-0]
              ELSE IF (DEST[63-0] == SNaN) THEN SRC[63-0];
              ELSE IF SRC[63-0] == SNaN) THEN SRC[63-0];
              ELSE IF (DEST[63-0] > SRC[63-0])
                THEN DEST[63-0]
                ELSE SRC[63-0];
              FI;
DEST[127-64] ← IF ((DEST[127-64] == 0.0) AND (SRC[127-64] == 0.0))
                THEN SRC[127-64]
                ELSE IF (DEST[127-64] == SNaN) THEN SRC[127-64];
                ELSE IF SRC[127-64] == SNaN) THEN SRC[127-64];
                ELSE IF (DEST[127-64] > SRC[63-0])
                  THEN DEST[127-64]
                  ELSE SRC[127-64];
                FI;

```

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

```
__m128d _mm_max_pd(__m128d a, __m128d b)
```

## MAXPD—Return Maximum Packed Double-Precision Floating-Point Values (Continued)

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.



## MAXPD—Return Maximum Packed Double-Precision Floating-Point Values (Continued)

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

## MAXPS—Return Maximum Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 5F /r	MAXPS <i>xmm1</i> , <i>xmm2/m128</i>	Return the maximum single-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a SIMD compare of the packed single-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of values to the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

If the values being compared are both 0.0s, the value in the second operand (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 operand (source operand), either a NaN or a valid floating-point value, is written to the result. This behavior allows compilers to use the MAXPS instruction for common C conditional constructs. 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 the MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

### Operation

```
DEST[31-0] ← IF ((DEST[31-0] == 0.0) AND (SRC[31-0] == 0.0)) THEN SRC[31-0]
              ELSE IF (DEST[31-0] == SNaN) THEN SRC[31-0];
              ELSE IF SRC[31-0] == SNaN) THEN SRC[31-0];
              ELSE IF (DEST[31-0] > SRC[31-0])
                THEN DEST[31-0]
                ELSE SRC[31-0];
```

FI;

\* repeat operation for 2nd and 3rd doublewords \*;

```
DEST[127-64] ← IF ((DEST[127-96] == 0.0) AND (SRC[127-96] == 0.0))
                THEN SRC[127-96]
                ELSE IF (DEST[127-96] == SNaN) THEN SRC[127-96];
                ELSE IF SRC[127-96] == SNaN) THEN SRC[127-96];
                ELSE IF (DEST[127-96] > SRC[127-96])
                  THEN DEST[127-96]
                  ELSE SRC[127-96];
```

FI;

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

```
__m128d _mm_max_ps(__m128d a, __m128d b)
```

## MAXPS—Return Maximum Packed Single-Precision Floating-Point Values (Continued)

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## **MAXPS—Return Maximum Packed Single-Precision Floating-Point Values (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

## MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

Opcode	Instruction	Description
F2 0F 5F /r	MAXSD <i>xmm1</i> , <i>xmm2/m64</i>	Return the maximum scalar double-precision floating-point value between <i>xmm2/mem64</i> and <i>xmm1</i> .

### Description

Compares the low double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value to the low quadword of the destination 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 a memory operand, only 64 bits are accessed. The high quadword of the destination operand remains unchanged.

If the values being compared are both 0.0s, the value in the second operand (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 operand (source operand), either a NaN or a valid floating-point value, is written to the result. This behavior allows compilers to use the MAXSD instruction for common C conditional constructs. 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 the MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

### Operation

```
DEST[63-0] ← IF ((DEST[63-0] == 0.0) AND (SRC[63-0] == 0.0)) THEN SRC[63-0]
              IF (DEST[63-0] == SNaN) THEN SRC[63-0];
              ELSE IF SRC[63-0] == SNaN) THEN SRC[63-0];
              ELSE IF (DEST[63-0] > SRC[63-0])
                  THEN DEST[63-0]
                  ELSE SRC[63-0];
              FI;
```

\* DEST[127-64] is unchanged \*;

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

```
__m128d _mm_max_sd(__m128d a, __m128d b)
```

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

## MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value (Continued)

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

## MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value (Continued)

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

Opcode	Instruction	Description
F3 0F 5F /r	MAXSS <i>xmm1</i> , <i>xmm2/m32</i>	Return the maximum scalar single-precision floating-point value between <i>xmm2/mem32</i> and <i>xmm1</i> .

### Description

Compares the low single-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value to the low doubleword of the destination 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 a memory operand, only 32 bits are accessed. The three high-order doublewords of the destination operand remain unchanged.

If the values being compared are both 0.0s, the value in the second operand (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 operand (source operand), either a NaN or a valid floating-point value, is written to the result. This behavior allows compilers to use the MAXSS instruction for common C conditional constructs. 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 the MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

### Operation

```
DEST[63-0] ← IF ((DEST[31-0] == 0.0) AND (SRC[31-0] == 0.0)) THEN SRC[31-0]
              ELSE IF (DEST[31-0] == SNaN) THEN SRC[31-0];
              ELSE IF SRC[31-0] == SNaN) THEN SRC[31-0];
              ELSE IF (DEST[31-0] > SRC[31-0])
                  THEN DEST[31-0]
                  ELSE SRC[31-0];
              FI;
```

\* DEST[127-32] is unchanged \*;

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

```
__m128d _mm_max_ss(__m128d a, __m128d b)
```

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.



## MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value (Continued)

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

## MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value (Continued)

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## MFENCE—Memory Fence

Opcode	Instruction	Description
OF AE /6	MFENCE	Serializes load and store operations.

### 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 in program order the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible. The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any SFENCE and LFENCE instructions, and any serializing instructions (such as the CPUID instruction).

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.

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). The PREFETCH $h$  instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the MFENCE instruction is not ordered with respect to PREFETCH $h$  instructions or any other speculative fetching mechanism (that is, data could be speculative loaded into the cache just before, during, or after the execution of an MFENCE instruction).

### 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)

None.

## MINPD—Return Minimum Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 5D /r	MINPD <i>xmm1</i> , <i>xmm2/m128</i>	Return the minimum double-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a SIMD compare of the packed double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of values to the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

If the values being compared are both 0.0s, the value in the second operand (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 operand (source operand), either a NaN or a valid floating-point value, is written to the result. This behavior allows compilers to use the MINPD instruction for common C conditional constructs. 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 the MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

### Operation

```

DEST[63-0] ← IF ((DEST[63-0] == 0.0) AND (SRC[63-0] == 0.0)) THEN SRC[63-0]
              ELSE IF (DEST[63-0] == SNaN) THEN SRC[63-0];
              ELSE IF SRC[63-0] == SNaN) THEN SRC[63-0];
              ELSE IF (DEST[63-0] < SRC[63-0])
                THEN DEST[63-0]
                ELSE SRC[63-0];
              FI;
DEST[127-64] ← IF ((DEST[127-64] == 0.0) AND (SRC[127-64] == 0.0))
                THEN SRC[127-64]
                ELSE IF (DEST[127-64] == SNaN) THEN SRC[127-64];
                ELSE IF SRC[127-64] == SNaN) THEN SRC[127-64];
                ELSE IF (DEST[127-64] < SRC[63-0])
                  THEN DEST[127-64]
                  ELSE SRC[127-64];
                FI;

```

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

```
__m128d _mm_min_pd(__m128d a, __m128d b)
```

## MINPD—Return Minimum Packed Double-Precision Floating-Point Values (Continued)

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## **MINPD—Return Minimum Packed Double-Precision Floating-Point Values (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

## MINPS—Return Minimum Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 5D /r	MINPS <i>xmm1</i> , <i>xmm2/m128</i>	Return the minimum single-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a SIMD compare of the packed single-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of values to the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

If the values being compared are both 0.0s, the value in the second operand (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 operand (source operand), either a NaN or a valid floating-point value, is written to the result. This behavior allows compilers to use the MINPS instruction for common C conditional constructs. 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 the MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

### Operation

```
DEST[63-0] ← IF ((DEST[31-0] == 0.0) AND (SRC[31-0] == 0.0)) THEN SRC[31-0]
              ELSE IF (DEST[31-0] == SNaN) THEN SRC[31-0];
              ELSE IF SRC[31-0] == SNaN) THEN SRC[31-0];
              ELSE IF (DEST[31-0] > SRC[31-0])
                  THEN DEST[31-0]
                  ELSE SRC[31-0];
              FI;
```

\* repeat operation for 2nd and 3rd doublewords \*;

```
DEST[127-64] ← IF ((DEST[127-96] == 0.0) AND (SRC[127-96] == 0.0))
                THEN SRC[127-96]
                ELSE IF (DEST[127-96] == SNaN) THEN SRC[127-96];
                ELSE IF SRC[127-96] == SNaN) THEN SRC[127-96];
                ELSE IF (DEST[127-96] < SRC[127-96])
                    THEN DEST[127-96]
                    ELSE SRC[127-96];
                FI;
```

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

```
__m128d _mm_min_ps(__m128d a, __m128d b)
```

## MINPS—Return Minimum Packed Single-Precision Floating-Point Values (Continued)

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.



## **MINPS—Minimum Packed Single-Precision Floating-Point Values (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

## MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

Opcode	Instruction	Description
F2 0F 5D /r	MINSD <i>xmm1</i> , <i>xmm2/m64</i>	Return the minimum scalar double-precision floating-point value between <i>xmm2/mem64</i> and <i>xmm1</i> .

### Description

Compares the low double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the minimum value to the low quadword of the destination 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 a memory operand, only the 64 bits are accessed. The high quadword of the destination operand remains unchanged.

If the values being compared are both 0.0s, the value in the second operand (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 operand (source operand), either a NaN or a valid floating-point value, is written to the result. This behavior allows compilers to use the MINSD instruction for common C conditional constructs. 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 the MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

### Operation

```
DEST[63-0] ← IF ((DEST[63-0] == 0.0) AND (SRC[63-0] == 0.0)) THEN SRC[63-0]
              ELSE IF (DEST[63-0] == SNaN) THEN SRC[63-0];
              ELSE IF SRC[63-0] == SNaN) THEN SRC[63-0];
              ELSE IF (DEST[63-0] < SRC[63-0])
                  THEN DEST[63-0]
                  ELSE SRC[63-0];
              FI;
```

\* DEST[127-64] is unchanged \*;

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

```
__m128d _mm_min_sd(__m128d a, __m128d b)
```

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

## MINSD—Return Minimum Scalar Double-Precision Floating-Point Value (Continued)

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

## **MINSD—Return Minimum Scalar Double-Precision Floating-Point Value (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

#AC(0)                If alignment checking is enabled and an unaligned memory reference is made.

## MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode	Instruction	Description
F3 0F 5D /r	MINSS <i>xmm1</i> , <i>xmm2/m32</i>	Return the minimum scalar single-precision floating-point value between <i>xmm2/mem32</i> and <i>xmm1</i> .

### Description

Compares the low single-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the minimum value to the low doubleword of the destination 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 a memory operand, only 32 bits are accessed. The three high-order doublewords of the destination operand remain unchanged.

If the values being compared are both 0.0s, the value in the second operand (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 operand (source operand), either a NaN or a valid floating-point value, is written to the result. This behavior allows compilers to use the MINSD instruction for common C conditional constructs. 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 the MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

### Operation

```
DEST[63-0] ← IF ((DEST[31-0] == 0.0) AND (SRC[31-0] == 0.0)) THEN SRC[31-0]
              ELSE IF (DEST[31-0] == SNaN) THEN SRC[31-0];
              ELSE IF SRC[31-0] == SNaN) THEN SRC[31-0];
              ELSE IF (DEST[31-0] < SRC[31-0])
                  THEN DEST[31-0]
                  ELSE SRC[31-0];
              FI;
```

\* DEST[127-32] is unchanged \*;

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

```
__m128d _mm_min_ss(__m128d a, __m128d b)
```

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

## MINSS—Return Minimum Scalar Single-Precision Floating-Point Value (Continued)

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

## MINSS—Return Minimum Scalar Single-Precision Floating-Point Value (Continued)

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## MOV—Move

Opcode	Instruction	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8
89 /r	MOV r/m16,r16	Move r16 to r/m16
89 /r	MOV r/m32,r32	Move r32 to r/m32
8A /r	MOV r8,r/m8	Move r/m8 to r8
8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg:offset)
A3	MOV moffs16*,AX	Move AX to (seg:offset)
A3	MOV moffs32*,EAX	Move EAX to (seg:offset)
B0+ rb	MOV r8,imm8	Move imm8 to r8
B8+ rw	MOV r16,imm16	Move imm16 to r16
B8+ rd	MOV r32,imm32	Move imm32 to r32
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32

## NOTES:

\* The *moffs8*, *moffs16*, and *moffs32* operands specify a simple offset relative to the segment base, where 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 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).

## 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, or a doubleword.

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.



## MOV—Move (Continued)

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>. 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 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 Pro processor, the two high-order bytes are filled with zeros; for earlier 32-bit IA-32 processors, the two high order bytes are undefined.

### 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 it points to.

IF SS is loaded;

---

1. Note that in a sequence of instructions that individually delay interrupts past the following instruction, only the first instruction in the sequence is guaranteed to delay the interrupt, but subsequent interrupt-delaying instructions may not delay the interrupt. Thus, in the following instruction sequence:

```
STI
MOV SS, EAX
MOV ESP, EBP
```

interrupts may be recognized before MOV ESP, EBP executes, because STI also delays interrupts for one instruction.

**MOV—Move (Continued)**

```

THEN
    IF segment selector is null
        THEN #GP(0);
    FI;
    IF segment selector index is outside descriptor table limits
        OR segment selector's RPL ≠ CPL
        OR segment is not a writable data segment
        OR DPL ≠ CPL
            THEN #GP(selector);
    FI;
    IF segment not marked present
        THEN #SS(selector);
ELSE
    SS ← segment selector;
    SS ← segment descriptor;
FI;
FI;
IF DS, ES, FS, or GS is loaded with non-null selector;
THEN
    IF segment selector index is outside descriptor table limits
        OR segment is not a data or readable code segment
        OR ((segment is a data or nonconforming code segment)
            AND (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 DS, ES, FS, or GS is loaded with a null selector;
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;

```

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)                    If attempt is made to load SS register with null segment selector.  
                           If the destination operand is in a nonwritable segment.

**MOV—Move (Continued)**

	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 nonwritable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If 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.

**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.

**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.

## MOV—Move (Continued)

- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If attempt is made to load the CS register.

## MOV—Move to/from Control Registers

Opcode	Instruction	Description
0F 22 /r	MOV CR0,r32	Move r32 to CR0
0F 22 /r	MOV CR2,r32	Move r32 to CR2
0F 22 /r	MOV CR3,r32	Move r32 to CR3
0F 22 /r	MOV CR4,r32	Move r32 to CR4
0F 20 /r	MOV r32,CR0	Move CR0 to r32
0F 20 /r	MOV r32,CR2	Move CR2 to r32
0F 20 /r	MOV r32,CR3	Move CR3 to r32
0F 20 /r	MOV r32,CR4	Move CR4 to r32

### Description

Moves the contents of a control register (CR0, CR2, CR3, or CR4) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits, regardless of the operand-size attribute. (See “Control Registers” in Chapter 2 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, 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.

When loading a control register, a program should not attempt to change any of the reserved bits; that is, always set reserved bits to the value previously read.

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 always 11B. The *r/m* field specifies the general-purpose register loaded or read.

These instructions have the following side effects:

- When writing to control register CR3, all non-global TLB entries are flushed (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*).

The following side effects are implementation specific for the Pentium Pro processors. Software should not depend on this functionality in future and previous IA-32 processors.:

- When modifying any of the paging flags in the control registers (PE and PG in register CR0 and PGE, PSE, and PAE in register CR4), all TLB entries are flushed, including global entries.
- If the PG flag is set to 1 and control register CR4 is written to set the PAE flag to 1 (to enable the physical address extension mode), the pointers (PDPTRs) in the page-directory pointers table will be loaded into the processor (into internal, non-architectural registers).
- If the PAE flag is set to 1 and the PG flag set to 1, writing to control register CR3 will cause the PDPTRs to be reloaded into the processor.
- If the PAE flag is set to 1 and control register CR0 is written to set the PG flag, the PDPTRs are reloaded into the processor.

## MOV—Move to/from Control Registers (Continued)

### Operation

DEST ← SRC;

### Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

### Protected Mode Exceptions

- #GP(0)                    If the current privilege level is not 0.
- If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
- If an attempt is made to write a 1 to any reserved bit in CR4.

### Real-Address Mode Exceptions

- #GP                        If an attempt is made to write a 1 to any reserved bit in CR4.

### Virtual-8086 Mode Exceptions

- #GP(0)                    These instructions cannot be executed in virtual-8086 mode.

## MOV—Move to/from Debug Registers

Opcode	Instruction	Description
0F 21/r	MOV r32, DR0-DR7	Move debug register to r32
0F 23 /r	MOV DR0-DR7,r32	Move r32 to debug register

### 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, regardless of the operand-size attribute. (See Chapter 14, *Debugging and Performance Monitoring*, of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, 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 set 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 always 11. The *r/m* field specifies the general-purpose register loaded or read.

### Operation

```
IF ((DE ← 1) and (SRC or DEST ← DR4 or DR5))
THEN
    #UD;
ELSE
    DEST ← SRC;
```

### Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the DE (debug extensions) bit of CR4 is set and a MOV instruction is executed involving DR4 or DR5.
#DB	If any debug register is accessed while the GD flag in debug register DR7 is set.

## MOV—Move to/from Debug Registers (Continued)

### Real-Address Mode Exceptions

- #UD                    If the DE (debug extensions) bit of CR4 is set and a MOV instruction is executed involving DR4 or DR5.
- #DB                    If any debug register is accessed while the GD flag in debug register DR7 is set.

### Virtual-8086 Mode Exceptions

- #GP(0)                The debug registers cannot be loaded or read when in virtual-8086 mode.



## MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 28 /r	MOVAPD <i>xmm1</i> , <i>xmm2/m128</i>	Move packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 29 /r	MOVAPD <i>xmm2/m128</i> , <i>xmm1</i>	Move packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .

### Description

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, 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 double-precision floating-point values to and from unaligned memory locations, use the MOVUPD instruction.

### Operation

DEST ← SRC;

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

```
__m128 _mm_load_pd(double * p)
void _mm_store_pd(double *p, __m128 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.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.

## MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values (Continued)

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 28 /r	MOVAPS <i>xmm1</i> , <i>xmm2/m128</i>	Move packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
0F 29 /r	MOVAPS <i>xmm2/m128</i> , <i>xmm1</i>	Move packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .

### Description

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, 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) is generated.

To move packed single-precision floating-point values to or from unaligned memory locations, use the MOVUPS instruction.

### Operation

DEST ← SRC;

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

`__m128 _mm_load_ps (float * p)`

`void _mm_store_ps (float *p, __m128 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.

If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0) For an illegal address in the SS segment.

#PF(fault-code) For a page fault.

#NM If TS in CR0 is set.

## MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values (Continued)

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVD—Move Doubleword

Opcode	Instruction	Description
0F 6E /r	MOVD <i>mm</i> , <i>r/m32</i>	Move doubleword from <i>r/m32</i> to <i>mm</i> .
0F 7E /r	MOVD <i>r/m32</i> , <i>mm</i>	Move doubleword from <i>mm</i> to <i>r/m32</i> .
66 0F 6E /r	MOVD <i>xmm</i> , <i>r/m32</i>	Move doubleword from <i>r/m32</i> to <i>xmm</i> .
66 0F 7E /r	MOVD <i>r/m32</i> , <i>xmm</i>	Move doubleword from <i>xmm</i> register to <i>r/m32</i> .

### 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 registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword an MMX 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 registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX 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.

### Operation

MOVD instruction when destination operand is MMX register:

```
DEST[31-0] ← SRC;
DEST[63-32] ← 00000000H;
```

MOVD instruction when destination operand is XMM register:

```
DEST[31-0] ← SRC;
DEST[127-32] ← 000000000000000000000000H;
```

MOVD instruction when source operand is MMX or XMM register:

```
DEST ← SRC[31-0];
```

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

```
MOVD      __m64 _mm_cvtsi32_si64 (int i)
MOVD      int _mm_cvtsi64_si32 ( __m64m )
MOVD      __m128i _mm_cvtsi32_si128 (int a)
MOVD      int _mm_cvtsi128_si32 ( __m128i a)
```

### Flags Affected

None.

## MOVD—Move Doubleword (Continued)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	If the destination operand is in a nonwritable segment. 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 EM in CR0 is set. (XMM register operations only.) If OSFXSR in CR4 is 0. (XMM register operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(MMX register operations only.) If there is a pending 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.

### Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set. (XMM register operations only.) If OSFXSR in CR4 is 0. (XMM register operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(MMX register operations only.) If there is a pending FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## MOVDQA—Move Aligned Double Quadword

Opcode	Instruction	Description
66 0F 6F /r	MOVDQA <i>xmm1</i> , <i>xmm2/m128</i>	Move aligned double quadword from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 7F /r	MOVDQA <i>xmm2/m128</i> , <i>xmm1</i>	Move aligned double quadword from <i>xmm1</i> to <i>xmm2/m128</i> .

### Description

Moves a double quadword 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 a double quadword to or from unaligned memory locations, use the MOVDQU instruction.

### Operation

DEST ← SRC;

\* #GP if SRC or DEST unaligned memory operand \*;

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

MOVDQA      `__m128i _mm_load_si128 ( __m128i *p)`

MOVDQA      `void _mm_store_si128 ( __m128i *p, __m128i a)`

### 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 segment limit.

If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)      If a memory operand effective address is outside the SS segment limit.

#NM          If TS in CR0 is set.

## MOVDQA—Move Aligned Double Quadword (Continued)

### Real-Address Mode Exceptions

#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#PF(fault-code)	If a page fault occurs.
#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------



## MOVDQU—Move Unaligned Double Quadword

Opcode	Instruction	Description
F3 0F 6F /r	MOVDQU <i>xmm1</i> , <i>xmm2/m128</i>	Move unaligned double quadword from <i>xmm2/m128</i> to <i>xmm1</i> .
F3 0F 7F /r	MOVDQU <i>xmm2/m128</i> , <i>xmm1</i>	Move unaligned double quadword from <i>xmm1</i> to <i>xmm2/m128</i> .

### Description

Moves a double quadword 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.

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.

### Operation

DEST ← SRC;

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

MOVDQU      void \_\_mm\_storeu\_si128 ( \_\_m128i \*p, \_\_m128i a)

MOVDQU      \_\_m128i \_\_mm\_loadu\_si128 ( \_\_m128i \*p)

### 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 segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	If TS in CR0 is set.

**MOVDQU—Move Unaligned Double Quadword (Continued)**

#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#PF(fault-code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVDQ2Q—Move Quadword from XMM to MMX Register

Opcode	Instruction	Description
F2 0F D6	MOVDQ2Q <i>mm, xmm</i>	Move low quadword from <i>xmm</i> to <i>mmx</i> register .

### 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 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.

### 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 TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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

## MOVHLPS— Move Packed Single-Precision Floating-Point Values High to Low

Opcode	Instruction	Description
OF 12 /r	MOVHLPS <i>xmm1</i> , <i>xmm2</i>	Move two packed single-precision floating-point values from high quadword of <i>xmm2</i> to low quadword of <i>xmm1</i> .

### Description

Moves two packed single-precision floating-point values from the high quadword of the source operand (second operand) to the low quadword of the destination operand (first operand). The high quadword of the destination operand is left unchanged.

### Operation

DEST[63-0] ← SRC[127-64];  
 \* DEST[127-64] unchanged \*;

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

MOVHLPS     \_\_m128 \_\_mm\_movehl\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM            If TS in CR0 is set.  
 #UD            If EM in CR0 is set.  
                   If OSFXSR in CR4 is 0.  
                   If CPUID feature flag SSE is 0.

### Real Address Mode Exceptions

Same exceptions as in Protected Mode.

### Virtual 8086 Mode Exceptions

Same exceptions as in Protected Mode.

## MOVHPD—Move High Packed Double-Precision Floating-Point Value

Opcode	Instruction	Description
66 0F 16 /r	MOVHPD <i>xmm</i> , <i>m64</i>	Move double-precision floating-point value from <i>m64</i> to high quadword of <i>xmm</i> .
66 0F 17 /r	MOVHPD <i>m64</i> , <i>xmm</i>	Move double-precision floating-point value from high quadword of <i>xmm</i> to <i>m64</i> .

### Description

Moves a double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory location. This instruction allows a double-precision floating-point value to be moved to and from the high quadword of an XMM register and memory. It cannot be used for register to register or memory to memory moves. When the destination operand is an XMM register, the low quadword of the register remains unchanged.

### Operation

MOVHPD instruction for memory to XMM move:

```
DEST[127-64] ← SRC ;
* DEST[63-0] unchanged *;
```

MOVHPD instruction for XMM to memory move:

```
DEST ← 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.

### 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.

#NM If TS in CR0 is set.

#UD If EM in CR0 is set.

## MOVHPD—Move High Packed Double-Precision Floating-Point Value (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#UD If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 16 /r	MOVHPS <i>xmm</i> , <i>m64</i>	Move two packed single-precision floating-point values from <i>m64</i> to high quadword of <i>xmm</i> .
0F 17 /r	MOVHPS <i>m64</i> , <i>xmm</i>	Move two packed single-precision floating-point values from high quadword of <i>xmm</i> to <i>m64</i> .

### Description

Moves two packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory location. This instruction allows two single-precision floating-point values to be moved to and from the high quadword of an XMM register and memory. It cannot be used for register to register or memory to memory moves. When the destination operand is an XMM register, the low quadword of the register remains unchanged.

### Operation

MOVHPD instruction for memory to XMM move:

```
DEST[127-64] ← SRC ;
* DEST[63-0] unchanged *;
```

MOVHPD instruction for XMM to memory move:

```
DEST ← 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.

### 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.
#NM	If TS in CR0 is set.

## MOVHPS—Move High Packed Single-Precision Floating-Point Values (Continued)

#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.



## MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode	Instruction	Description
OF 16 /r	MOVLHPS <i>xmm1</i> , <i>xmm2</i>	Move two packed single-precision floating-point values from low quadword of <i>xmm2</i> to high quadword of <i>xmm1</i> .

### Description

Moves two packed single-precision floating-point values from the low quadword of the source operand (second operand) to the high quadword of the destination operand (first operand). The high quadword of the destination operand is left unchanged.

### Operation

DEST[127-64] ← SRC[63-0];  
 \* DEST[63-0] unchanged \*;

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

MOVHLPS     \_\_m128 \_\_mm\_movelh\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM            If TS in CR0 is set.  
 #UD            If EM in CR0 is set.  
                   If OSFXSR in CR4 is 0.  
                   If CPUID feature flag SSE is 0.

### Real Address Mode Exceptions

Same exceptions as in Protected Mode.

### Virtual 8086 Mode Exceptions

Same exceptions as in Protected Mode.

## MOVLPD—Move Low Packed Double-Precision Floating-Point Value

Opcode	Instruction	Description
66 0F 12 /r	MOVLPD <i>xmm</i> , <i>m64</i>	Move double-precision floating-point value from <i>m64</i> to low quadword of <i>xmm</i> register.
66 0F 13 /r	MOVLPD <i>m64</i> , <i>xmm</i>	Move double-precision floating-point nvalue from low quadword of <i>xmm</i> register to <i>m64</i> .

### Description

Moves a double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory location. This instruction allows a double-precision floating-point value to be moved to and from the low quadword of an XMM register and memory. It cannot be used for register to register or memory to memory moves. When the destination operand is an XMM register, the high quadword of the register remains unchanged.

### Operation

MOVLPD instruction for memory to XMM move:

```
DEST[63-0] ← SRC ;
* DEST[127-64] unchanged *;
```

MOVLPD instruction for XMM to memory move:

```
DEST ← 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.

### 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.
#NM	If TS in CR0 is set.

## MOVLPD—Move Low Packed Double-Precision Floating-Point Value (Continued)

#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 12 /r	MOVLPS <i>xmm</i> , <i>m64</i>	Move two packed single-precision floating-point values from <i>m64</i> to low quadword of <i>xmm</i> .
0F 13 /r	MOVLPS <i>m64</i> , <i>xmm</i>	Move two packed single-precision floating-point values from low quadword of <i>xmm</i> to <i>m64</i> .

### Description

Moves two packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory location. This instruction allows two single-precision floating-point values to be moved to and from the low quadword of an XMM register and memory. It cannot be used for register to register or memory to memory moves. When the destination operand is an XMM register, the high quadword of the register remains unchanged.

### Operation

MOVLPS instruction for memory to XMM move:

```
DEST[63-0] ← SRC ;
* DEST[127-64] unchanged *;
```

MOVLPS instruction for XMM to memory move:

```
DEST ← SRC[63-0] ;
```

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

```
MOVLPS    __m128 _mm_loadl_pi ( __m128 a, __m64 *p)
```

```
MOVLPS    void _mm_storel_pi (__m64 *p, __m128 a)
```

### SIMD Floating-Point Exceptions

None.

### 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.
- #NM If TS in CR0 is set.

## MOVLPS—Move Low Packed Single-Precision Floating-Point Values (Continued)

#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

Opcode	Instruction	Description
66 0F 50 /r	MOVMSKPD <i>r32</i> , <i>xmm</i>	Extract 2-bit sign mask of from <i>xmm</i> and store in <i>r32</i> .

### 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.

### Operation

```
DEST[0] ← SRC[63];
DEST[1] ← SRC[127];
DEST[3-2] ← 00B;
DEST[31-4] ← 0000000H;
```

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

```
MOVMSKPD    int_mm_movemask_pd ( __m128 a)
```

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

Same exceptions as in Protected Mode

## **MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Protected Mode

## MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

Opcode	Instruction	Description
0F 50 /r	MOVMSKPS <i>r32</i> , <i>xmm</i>	Extract 4-bit sign mask of from <i>xmm</i> and store in <i>r32</i> .

### Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4-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 4 low-order bits of the destination operand.

### Operation

```
DEST[0] ← SRC[31];
DEST[1] ← SRC[63];
DEST[2] ← SRC[95];
DEST[3] ← SRC[127];
DEST[31-4] ← 000000H;
```

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

```
int_mm_movemask_ps(__m128 a)
```

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

Same exceptions as in Protected Mode



## **MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask (Continued)**

### **Virtual 8086 Mode Exceptions**

Same exceptions as in Protected Mode.

## MOVNTDQ—Store Double Quadword Using Non-Temporal Hint

Opcode	Instruction	Description
66 0F E7 /r	MOVNTDQ <i>m128</i> , <i>xmm</i>	Move double quadword from <i>xmm</i> to <i>m128</i> using non-temporal hint.

### Description

Moves the double quadword 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, which is assumed to contain integer data (packed bytes, words, doublewords, or quadwords). The destination operand is a 128-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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

### Operation

DEST ← SRC;

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

MOVNTDQ      void\_mm\_stream\_si128 ( \_\_m128i \*p, \_\_m128i 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.<br><br>If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment.   |

## MOVNTDQ—Store Double Quadword Using Non-Temporal Hint (Continued)

#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode	Instruction	Description
0F C3 /r	MOVNI <i>m32, r32</i>	Move doubleword from <i>r32</i> to <i>m32</i> using non-temporal hint.

### 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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

### Operation

DEST ← SRC;

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

MOVNTDQ     void\_mm\_stream\_si32 (int \*p, int 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 feature flag SSE2 is 0.

## MOVNTI—Store Doubleword Using Non-Temporal Hint (Continued)

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

Opcode	Instruction	Description
66 0F 2B /r	MOVNTPD <i>m128, xmm</i>	Move packed double-precision floating-point values from <i>xmm</i> to <i>m128</i> using non-temporal hint.

### Description

Moves the double 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 XMM register, which is assumed to contain two packed double-precision floating-point values. The destination operand is a 128-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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

### Operation

DEST ← SRC;

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

MOVNTDQ     void\_mm\_stream\_pd(double \*p, \_\_m128i 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.
- If memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #SS(0)     For an illegal address in the SS segment.

## MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint (Continued)

#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode	Instruction	Description
0F 2B /r	MOVNTPS <i>m128, xmm</i>	Move packed single-precision floating-point values from <i>xmm</i> to <i>m128</i> using non-temporal hint.

### Description

Moves the double 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 XMM register, which is assumed to contain four packed single-precision floating-point values. The destination operand is a 128-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 *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

### Operation

DEST ← SRC;

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

MOVNTDQ      void\_mm\_stream\_ps(float \* p, \_\_m128 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.



## MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint (Continued)

	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Description
0F E7 /r	MOVNTQ <i>m64, mm</i>	Move quadword from <i>mm</i> to <i>m64</i> using non-temporal hint.

### 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 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

### Operation

DEST ← SRC;

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

MOVNTQ      void\_mm\_stream\_pi(\_\_m64 \* p, \_\_m64 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.
#NM	If TS in CR0 is set.

## MOVNTQ—Store of Quadword Using Non-Temporal Hint (Continued)

#MF	If there is a pending x87 FPU exception.
#UD	If EM in CR0 is set. If CPUID feature flag SSE is 0.
#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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#UD	If EM in CR0 is set. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## MOVQ—Move Quadword

Opcode	Instruction	Description
0F 6F /r	MOVQ <i>mm</i> , <i>mm/m64</i>	Move quadword from <i>mm/m64</i> to <i>mm</i> .
0F 7F /r	MOVQ <i>mm/m64</i> , <i>mm</i>	Move quadword from <i>mm</i> to <i>mm/m64</i> .
F3 0F 7E	MOVQ <i>xmm1</i> , <i>xmm2/m64</i>	Move quadword from <i>xmm2/mem64</i> to <i>xmm1</i> .
66 0F D6	MOVQ <i>xmm2/m64</i> , <i>xmm1</i>	Move quadword from <i>xmm1</i> to <i>xmm2/mem64</i> .

### Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX registers or between an MMX 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.

### Operation

MOVQ instruction when operating on MMX registers and memory locations:

DEST ← SRC;

MOVQ instruction when source and destination operands are XMM registers:

DEST[63-0] ← SRC[63-0];

MOVQ instruction when source operand is XMM register and destination operand is memory location:

DEST ← SRC[63-0];

MOVQ instruction when source operand is memory location and destination operand is XMM register:

DEST[63-0] ← SRC;

DEST[127-64] ← 0000000000000000H;

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

## MOVQ—Move Quadword (Continued)

### Protected Mode Exceptions

#GP(0)	If the destination operand is in a nonwritable segment. 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 EM in CR0 is set. (XMM register operations only.) If OSFXSR in CR4 is 0. (XMM register operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(MMX register operations only.) If there is a pending 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.

### Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set. (XMM register operations only.) If OSFXSR in CR4 is 0. (XMM register operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(MMX register operations only.) If there is a pending FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

**MOVQ2DQ—Move Quadword from MMX to XMM Register**

Opcode	Instruction	Description
F3 0F D6	MOVQ2DQ <i>xmm, mm</i>	Move quadword from <i>mmx</i> to low quadword of <i>xmm</i> .

**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 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.

**Operation**

DEST[63-0] ← SRC[63-0];  
 DEST[127-64] ← 000000000000000000H;

**intel C/C++ Compiler Intrinsic Equivalent**

MOVQ2DQ     \_\_128i\_mm\_movpi64\_pi64 ( \_\_m64 a)

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM            If TS in CR0 is set.  
 #UD            If EM in CR0 is set.  
               If OSFXSR in CR4 is 0.  
               If CPUID feature flag SSE2 is 0.  
 #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

## MOVS/MOVSb/MOVSW/MOVSd—Move Data from String to String

Opcode	Instruction	Description
A4	MOVS m8, m8	Move byte at address DS:(E)SI to address ES:(E)DI
A5	MOVS m16, m16	Move word at address DS:(E)SI to address ES:(E)DI
A5	MOVS m32, m32	Move doubleword at address DS:(E)SI to address ES:(E)DI
A4	MOVSb	Move byte at address DS:(E)SI to address ES:(E)DI
A5	MOVSW	Move word at address DS:(E)SI to address ES:(E)DI
A5	MOVSd	Move doubleword at address DS:(E)SI to address ES:(E)DI

### 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 incremented; 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.

The MOVS, MOVSb, MOVSW, and MOVSd instructions can be preceded by the REP prefix (see “REP/REPE/REPZ/REPNE/REPnz—Repeat String Operation Prefix” in this chapter) for block moves of ECX bytes, words, or doublewords.

## MOVS/MOVSMB/MOVSW/MOVSD—Move Data from String to String (Continued)

### Operation

```

DEST ← SRC;
IF (byte move)
  THEN IF DF ← 0
    THEN
      (E)SI ← (E)SI + 1;
      (E)DI ← (E)DI + 1;
    ELSE
      (E)SI ← (E)SI - 1;
      (E)DI ← (E)DI - 1;
    FI;
  ELSE IF (word move)
    THEN IF DF ← 0
      (E)SI ← (E)SI + 2;
      (E)DI ← (E)DI + 2;
    ELSE
      (E)SI ← (E)SI - 2;
      (E)DI ← (E)DI - 2;
    FI;
  ELSE (* doubleword move*)
    THEN IF DF ← 0
      (E)SI ← (E)SI + 4;
      (E)DI ← (E)DI + 4;
    ELSE
      (E)SI ← (E)SI - 4;
      (E)DI ← (E)DI - 4;
    FI;
  FI;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

- |                 |   |
|-----------------|---|
| #GP(0)          | If the destination is located in a nonwritable segment.<br>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 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.   |



## MOVSB/MOVSQ/MOVSQ/MOVSD—Move Data from String to String (Continued)

#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.

### 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.

## MOVSD—Move Scalar Double-Precision Floating-Point Value

Opcode	Instruction	Description
F2 0F 10 /r	MOVSD <i>xmm1</i> , <i>xmm2/m64</i>	Move scalar double-precision floating-point value from <i>xmm2/m64</i> to <i>xmm1</i> register.
F2 0F 11 /r	MOVSD <i>xmm2/m64</i> , <i>xmm</i>	Move scalar double-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m64</i> .

### Description

Moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

When the source and destination operands are XMM registers, the high quadword of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high quadword of the destination operand is cleared to all 0s.

### Operation

MOVSD instruction when source and destination operands are XMM registers:

```
DEST[63-0] ← SRC[63-0];
* DEST[127-64] remains unchanged *;
```

MOVSD instruction when source operand is XMM register and destination operand is memory location:

```
DEST ← SRC[63-0];
```

MOVSD instruction when source operand is memory location and destination operand is XMM register:

```
DEST[63-0] ← SRC;
DEST[127-64] ← 0000000000000000H;
```

### 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.

## MOVSD—Move Scalar Double-Precision Floating-Point Value (Continued)

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## **MOVSD—Move Scalar Double-Precision Floating-Point Value (Continued)**

#AC(0)

If alignment checking is enabled and an unaligned memory reference is made.

## MOVSS—Move Scalar Single-Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 10 /r	MOVSS <i>xmm1</i> , <i>xmm2/m32</i>	Move scalar single-precision floating-point value from <i>xmm2/m64</i> to <i>xmm1</i> register.
F3 0F 11 /r	MOVSS <i>xmm2/m32</i> , <i>xmm</i>	Move scalar single-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m64</i> .

### Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

When the source and destination operands are XMM registers, the three high-order doublewords of the destination operand remain unchanged. When the source operand is a memory location and destination operand is an XMM registers, the three high-order doublewords of the destination operand are cleared to all 0s.

### Operation

MOVSS instruction when source and destination operands are XMM registers:

```
DEST[31-0] ← SRC[31-0];
* DEST[127-32] remains unchanged *;
```

MOVSS instruction when source operand is XMM register and destination operand is memory location:

```
DEST ← SRC[31-0];
```

MOVSS instruction when source operand is memory location and destination operand is XMM register:

```
DEST[31-0] ← SRC;
DEST[127-32] ← 000000000000000000000000H;
```

### 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.

## MOVSS—Move Scalar Single-Precision Floating-Point Value (Continued)

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## **MOVSS—Move Scalar Single-Precision Floating-Point Value (Continued)**

#AC(0)                      If alignment checking is enabled and an unaligned memory reference is made.

## MOVSX—Move with Sign-Extension

Opcode	Instruction	Description
0F BE /r	MOVSX r16,r/m8	Move byte to word with sign-extension
0F BE /r	MOVSX r32,r/m8	Move byte to doubleword, sign-extension
0F BF /r	MOVSX r32,r/m16	Move word to doubleword, sign-extension

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

### 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.

### 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.



**MOVSX—Move with Sign-Extension (Continued)**

#PF(fault-code)      If a page fault occurs.

## MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 10 /r	MOVUPD <i>xmm1</i> , <i>xmm2/m128</i>	Move packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 11 /r	MOVUPD <i>xmm2/m128</i> , <i>xmm</i>	Move packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .

### Description

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, to store the contents of an XMM register into a 128-bit memory location, or 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.

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.

### Operation

DEST ← SRC;

\* #GP if SRC or DEST unaligned memory operand \*;

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

MOVUPD     \_\_m128 \_mm\_loadu\_pd(double \* p)

MOVUPD     void \_mm\_storeu\_pd(double \*p, \_\_m128 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.

## MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values (Continued)

#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 10 /r	MOVUPS <i>xmm1, xmm2/m128</i>	Move packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
0F 11 /r	MOVUPS <i>xmm2/m128, xmm1</i>	Move packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .

### Description

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, to store the contents of an XMM register into a 128-bit memory location, or 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.

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.

### Operation

DEST ← SRC;

\* #GP if SRC or DEST unaligned memory operand \*;

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

MOVUPS        `__m128 _mm_loadu_ps(double * p)`

MOVUPS        `void _mm_storeu_ps(double *p, __m128 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.

## MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values (Continued)

#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MOVZX—Move with Zero-Extend

Opcode	Instruction	Description
0F B6 /r	MOVZX <i>r16,r/m8</i>	Move byte to word with zero-extension
0F B6 /r	MOVZX <i>r32,r/m8</i>	Move byte to doubleword, zero-extension
0F B7 /r	MOVZX <i>r32,r/m16</i>	Move word to doubleword, zero-extension

### Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

### 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.

### 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.

**MOVZX—Move with Zero-Extend (Continued)**

#AC(0)                    If alignment checking is enabled and an unaligned memory reference is made.

## MUL—Unsigned Multiply

Opcode	Instruction	Description
F6 /4	MUL <i>r/m8</i>	Unsigned multiply ( $AX \leftarrow AL * r/m8$ )
F7 /4	MUL <i>r/m16</i>	Unsigned multiply ( $DX:AX \leftarrow AX * r/m16$ )
F7 /4	MUL <i>r/m32</i>	Unsigned multiply ( $EDX:EAX \leftarrow EAX * r/m32$ )

### 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 the following table.

Operand Size	Source 1	Source 2	Destination
Byte	AL	<i>r/m8</i>	AX
Word	AX	<i>r/m16</i>	DX:AX
Dword	EAX	<i>r/m32</i>	EDX:EAX

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.

### Operation

```

IF byte operation
  THEN
    AX ← AL * SRC
  ELSE (* word or doubleword operation *)
    IF OperandSize ← 16
      THEN
        DX:AX ← AX * SRC
      ELSE (* OperandSize ← 32 *)
        EDX:EAX ← EAX * SRC
    FI;
  FI;

```

### Flags Affected

The OF and CF flags are cleared 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.



## MUL—Unsigned Multiply (Continued)

### 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.

### 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.

## MULPD—Multiply Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 59 /r	MULPD <i>xmm1</i> , <i>xmm2/m128</i>	Multiply packed double-precision floating-point values in <i>xmm2/m128</i> by <i>xmm1</i> .

### Description

Performs a SIMD multiply 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. 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD double-precision floating-point operation.

### Operation

```
DEST[63-0] ← DEST[63-0] * SRC[63-0];
DEST[127-64] ← DEST[127-64] * SRC[127-64];
```

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

```
MULPD      __m128d __mm_mul_pd (m128d a, m128d b)
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.

## MULPD—Multiply Packed Double-Precision Floating-Point Values (Continued)

If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 59 /r	MULPS <i>xmm1</i> , <i>xmm2/m128</i>	Multiply packed single-precision floating-point values in <i>xmm2/mem</i> by <i>xmm1</i> .

### 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. 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD single-precision floating-point operation.

### Operation

```
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];
```

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

```
MULPS      __m128 _mm_mul_ps(__m128 a, __m128 b)
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

## MULPS—Multiply Packed Single-Precision Floating-Point Values (Continued)

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## MULSD—Multiply Scalar Double-Precision Floating-Point Values

Opcode	Instruction	Description
F2 0F 59 /r	MULSD <i>xmm1</i> , <i>xmm2/m64</i>	Multiply the low double-precision floating-point value in <i>xmm2/mem64</i> by low double-precision floating-point value in <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

### Operation

DEST[63-0] ← DEST[63-0] \* xmm2/m64[63-0];  
 \* DEST[127-64] remains unchanged \*;

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

MULSD        `__m128d __mm_mul_sd (m128d a, m128d b)`

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0.

## MULSD—Multiply Scalar Double-Precision Floating-Point Values (Continued)

If CPUID feature flag SSE2 is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC For unaligned memory reference if the current privilege level is 3.

## MULSS—Multiply Scalar Single-Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 59 /r	MULSS <i>xmm1</i> , <i>xmm2/m32</i>	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> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

### Operation

DEST[31-0] ← DEST[31-0] \* SRC[31-0];

\* DEST[127-32] remains unchanged \*;

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

MULSS            \_\_m128 \_mm\_mul\_ss(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.



## MULSS—Multiply Scalar Single-Precision Floating-Point Values (Continued)

If CPUID feature flag SSE is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC For unaligned memory reference if the current privilege level is 3.

## NEG—Two's Complement Negation

Opcode	Instruction	Description
F6 /3	NEG <i>r/m8</i>	Two's complement negate <i>r/m8</i>
F7 /3	NEG <i>r/m16</i>	Two's complement negate <i>r/m16</i>
F7 /3	NEG <i>r/m32</i>	Two's complement negate <i>r/m32</i>

### 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.

### Operation

```
IF DEST ← 0
  THEN CF ← 0
  ELSE CF ← 1;
FI;
DEST ← -(DEST)
```

### Flags Affected

The CF flag cleared 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 nonwritable 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.

## NEG—Two's Complement Negation (Continued)

### 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.

## NOP—No Operation

Opcode	Instruction	Description
90	NOP	No operation

### Description

Performs no operation. This instruction is a one-byte instruction that takes up space in the instruction stream but does not affect the machine context, except the EIP register.

The NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

### Flags Affected

None.

### Exceptions (All Operating Modes)

None.

## NOT—One's Complement Negation

Opcode	Instruction	Description
F6 /2	NOT <i>r/m8</i>	Reverse each bit of <i>r/m8</i>
F7 /2	NOT <i>r/m16</i>	Reverse each bit of <i>r/m16</i>
F7 /2	NOT <i>r/m32</i>	Reverse each bit of <i>r/m32</i>

### Description

Performs a bitwise NOT operation (each 1 is cleared 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.

### Operation

DEST ← NOT DEST;

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the destination operand points to a nonwritable 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

#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.

## NOT—One's Complement Negation (Continued)

### 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.

## OR—Logical Inclusive OR

Opcode	Instruction	Description
0C <i>ib</i>	OR AL, <i>imm8</i>	AL OR <i>imm8</i>
0D <i>iw</i>	OR AX, <i>imm16</i>	AX OR <i>imm16</i>
0D <i>id</i>	OR EAX, <i>imm32</i>	EAX OR <i>imm32</i>
80 /1 <i>ib</i>	OR <i>r/m8</i> , <i>imm8</i>	<i>r/m8</i> OR <i>imm8</i>
81 /1 <i>iw</i>	OR <i>r/m16</i> , <i>imm16</i>	<i>r/m16</i> OR <i>imm16</i>
81 /1 <i>id</i>	OR <i>r/m32</i> , <i>imm32</i>	<i>r/m32</i> OR <i>imm32</i>
83 /1 <i>ib</i>	OR <i>r/m16</i> , <i>imm8</i>	<i>r/m16</i> OR <i>imm8</i> ( <i>sign-extended</i> )
83 /1 <i>ib</i>	OR <i>r/m32</i> , <i>imm8</i>	<i>r/m32</i> OR <i>imm8</i> ( <i>sign-extended</i> )
08 /r	OR <i>r/m8</i> , <i>r8</i>	<i>r/m8</i> OR <i>r8</i>
09 /r	OR <i>r/m16</i> , <i>r16</i>	<i>r/m16</i> OR <i>r16</i>
09 /r	OR <i>r/m32</i> , <i>r32</i>	<i>r/m32</i> OR <i>r32</i>
0A /r	OR <i>r8</i> , <i>r/m8</i>	<i>r8</i> OR <i>r/m8</i>
0B /r	OR <i>r16</i> , <i>r/m16</i>	<i>r16</i> OR <i>r/m16</i>
0B /r	OR <i>r32</i> , <i>r/m32</i>	<i>r32</i> OR <i>r/m32</i>

### 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.

### 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 nonwritable 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.

**OR—Logical Inclusive OR (Continued)**

#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**

#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.



## ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 56 /r	ORPD <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical 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.

### Operation

DEST[127:0] ← DEST[127:0] BitwiseOR SRC[127:0];

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

ORPD            \_\_m128d \_mm\_or\_pd(\_\_m128d a, \_\_m128d b)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## ORPD—Bitwise Logical OR of Packed Double-Precision Floating-Point Values (Continued)

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 56 /r	ORPS <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i>

### Description

Performs a bitwise logical 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.

### Operation

DEST[127:0] ← DEST[127:0] BitwiseOR SRC[127:0];

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

ORPS            \_\_m128 \_mm\_or\_ps(\_\_m128 a, \_\_m128 b)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.

## ORPS—Bitwise Logical OR of Packed Single-Precision Floating-Point Values (Continued)

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## OUT—Output to Port

Opcode	Instruction	Description
E6 <i>ib</i>	OUT <i>imm8</i> , AL	Output byte in AL to I/O port address <i>imm8</i>
E7 <i>ib</i>	OUT <i>imm8</i> , AX	Output word in AX to I/O port address <i>imm8</i>
E7 <i>ib</i>	OUT <i>imm8</i> , EAX	Output doubleword in EAX to I/O port address <i>imm8</i>
EE	OUT DX, AL	Output byte in AL to I/O port address in DX
EF	OUT DX, AX	Output word in AX to I/O port address in DX
EF	OUT DX, EAX	Output doubleword in EAX to I/O port address in DX

### 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 12, *Input/Output*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

### IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium processor insures 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; the other IA-32 processors do not.

### 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;
```

## OUT—Output to Port (Continued)

ELSE (Real Mode or Protected Mode with  $CPL \leq IOPL$  \*)

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.

### Real-Address Mode Exceptions

None.

### 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.

## OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode	Instruction	Description
6E	OUTS DX, m8	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTS DX, m16	Output word from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTS DX, m32	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX
6E	OUTSB	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTSW	Output word from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTSD	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX

### 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:EDI or the DS:DI 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.) 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 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 (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 (E)SI 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.

## OUTS/OUTSB/OUTSW/OUTSD—Output String to Port (Continued)

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/REP NZ—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 12, *Input/Output*, in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

### IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor insures 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; the other IA-32 processors do not. For the Pentium 4 and P6 family processors, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

### Operation

```

IF ((PE ← 1) AND ((CPL > IOPL) OR (VM ← 1)))
  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
    IF (Any I/O Permission Bit for I/O port being accessed ← 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
      ELSE (* I/O operation is allowed *)
        DEST ← SRC; (* Writes to I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Writes to I/O port *)
FI;
IF (byte transfer)
  THEN IF DF ← 0
    THEN (E)SI ← (E)SI + 1;
    ELSE (E)SI ← (E)SI - 1;
  FI;
  ELSE IF (word transfer)
    THEN IF DF ← 0
      THEN (E)SI ← (E)SI + 2;
      ELSE (E)SI ← (E)SI - 2;
    FI;
    ELSE (* doubleword transfer *)
      THEN IF DF ← 0
        THEN (E)SI ← (E)SI + 4;
        ELSE (E)SI ← (E)SI - 4;
      FI; FI; FI;

```



## OUTS/OUTSB/OUTSW/OUTSD—Output String to Port (Continued)

### 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.

### 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 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.

### PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode	Instruction	Description
0F 63 /r	PACKSSWB <i>mm1</i> , <i>mm2/m64</i>	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 0F 63 /r	PACKSSWB <i>xmm1</i> , <i>xmm2/m128</i>	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.
0F 6B /r	PACKSSDW <i>mm1</i> , <i>mm2/m64</i>	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 0F 6B /r	PACKSSDW <i>xmm1</i> , <i>xmm2/m128</i>	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.

#### 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 3-5 for an example of the packing operation.

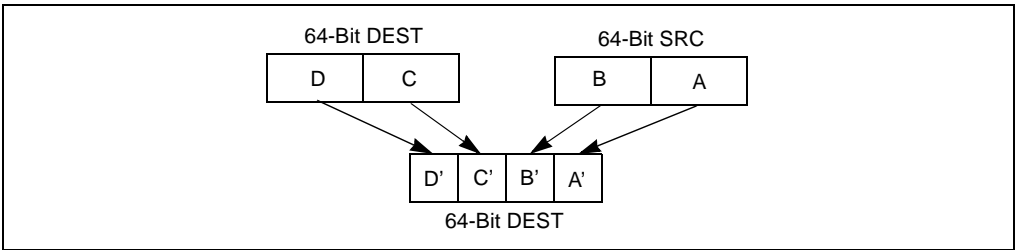


Figure 3-5. Operation of the PACKSSDW Instruction Using 64-bit Operands.

The PACKSSWB instruction converts 4 or 8 signed word integers from the destination operand (first operand) and 4 or 8 signed word integers from the source operand (second operand) into 8 or 16 signed byte integers and stores the result in the destination operand. If a signed word integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination.

## PACKSSWB/PACKSSDW—Pack with Signed Saturation (Continued)

The PACKSSDW instruction packs 2 or 4 signed doublewords from the destination operand (first operand) and 2 or 4 signed doublewords from the source operand (second operand) into 4 or 8 signed words in the destination operand (see Figure 3-5). 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 PACKSSWB and PACKSSDW instructions operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX register and the source operand can be either an MMX register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

### Operation

PACKSSWB instruction 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 instruction 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 with 128-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 (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]);
```

## PACKSSWB/PACKSSDW—Pack with Signed Saturation (Continued)

DEST[111-104] ← SaturateSignedWordToSignedByte (SRC[95-80]);  
 DEST[119-112] ← SaturateSignedWordToSignedByte (SRC[111-96]);  
 DEST[127-120] ← SaturateSignedWordToSignedByte (SRC[127-112]);

PACKSSDW instruction with 128-bit operands

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]);

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

`__m64 _mm_packs_pi16(__m64 m1, __m64 m2)`  
`__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)`

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PACKSSWB/PACKSSDW—Pack with Signed Saturation (Continued)

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

## PACKUSWB—Pack with Unsigned Saturation

Opcode	Instruction	Description
0F 67 /r	PACKUSWB <i>mm, mm/m64</i>	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 0F 67 /r	PACKUSWB <i>xmm1, xmm2/m128</i>	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.

### Description

Converts 4 or 8 signed word integers from the destination operand (first operand) and 4 or 8 signed word integers from the source operand (second operand) into 8 or 16 unsigned byte integers and stores the result in the destination operand. (See Figure 3-5 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 or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX register and the source operand can be either an MMX register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

### Operation

PACKUSWB instruction 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 instruction with 128-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 (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]);
```

## PACKUSWB—Pack with Unsigned Saturation (Continued)

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]);

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

`__m64 _mm_packs_pu16(__m64 m1, __m64 m2)`

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	<p>If EM in CR0 is set.</p> <p>(128-bit operations only.) If OSFXSR in CR4 is 0.</p> <p>(128-bit operations only.) If CPUID feature flag SSE2 is 0.</p>
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	<p>(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside of the effective address space from 0 to FFFFH.</p>
#UD	<p>If EM in CR0 is set.</p> <p>(128-bit operations only.) If OSFXSR in CR4 is 0.</p>

**PACKUSWB—Pack with Unsigned Saturation (Continued)**

(128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.



## PADDB/PADDW/PADDD—Add Packed Integers

Opcode	Instruction	Description
0F FC /r	PADDB <i>mm, mm/m64</i>	Add packed byte integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FC /r	PADDB <i>xmm1, xmm2/m128</i>	Add packed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> .
0F FD /r	PADDW <i>mm, mm/m64</i>	Add packed word integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FD /r	PADDW <i>xmm1, xmm2/m128</i>	Add packed word integers from <i>xmm2/m128</i> and <i>xmm1</i> .
0F FE /r	PADDD <i>mm, mm/m64</i>	Add packed doubleword integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FE /r	PADDD <i>xmm1, xmm2/m128</i>	Add packed doubleword integers from <i>xmm2/m128</i> and <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX register and the source operand can be either an MMX register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PADDB instruction adds packed byte integers. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW instruction adds packed word integers. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand.

The PADDD instruction adds packed doubleword integers. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand.

Note that the PADDB, PADDW, and PADDD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

## PADDB/PADDW/PADDD—Add Packed Integers (Continued)

### Operation

PADDB instruction with 64-bit operands:

$DEST[7..0] \leftarrow DEST[7..0] + SRC[7..0];$   
 \* repeat add operation for 2nd through 7th byte \*;  
 $DEST[63..56] \leftarrow DEST[63..56] + SRC[63..56];$

PADDB instruction with 128-bit operands:

$DEST[7-0] \leftarrow DEST[7-0] + SRC[7-0];$   
 \* repeat add operation for 2nd through 14th byte \*;  
 $DEST[127-120] \leftarrow DEST[111-120] + SRC[127-120];$

PADDW instruction with 64-bit operands:

$DEST[15..0] \leftarrow DEST[15..0] + SRC[15..0];$   
 \* repeat add operation for 2nd and 3th word \*;  
 $DEST[63..48] \leftarrow DEST[63..48] + SRC[63..48];$

PADDW instruction with 128-bit operands:

$DEST[15-0] \leftarrow DEST[15-0] + SRC[15-0];$   
 \* repeat add operation for 2nd through 7th word \*;  
 $DEST[127-112] \leftarrow DEST[127-112] + SRC[127-112];$

PADDD instruction with 64-bit operands:

$DEST[31..0] \leftarrow DEST[31..0] + SRC[31..0];$   
 $DEST[63..32] \leftarrow DEST[63..32] + SRC[63..32];$

PADDD instruction with 128-bit operands:

$DEST[31-0] \leftarrow DEST[31-0] + SRC[31-0];$   
 \* repeat add operation for 2nd and 3th doubleword \*;  
 $DEST[127-96] \leftarrow DEST[127-96] + SRC[127-96];$

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

PADDB `__m64 _mm_add_pi8(__m64 m1, __m64 m2)`  
 PADDB `__m128i _mm_add_epi8 (__m128ia, __m128ib )`  
 PADDW `__m64 _mm_addw_pi16(__m64 m1, __m64 m2)`  
 PADDW `__m128i _mm_add_epi16 ( __m128i a, __m128i b)`  
 PADDD `__m64 _mm_add_pi32(__m64 m1, __m64 m2)`  
 PADDD `__m128i _mm_add_epi32 ( __m128i a, __m128i b)`

### Flags Affected

None.

## PADDB/PADDW/PADDD—Add Packed Integers (Continued)

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

## PADDQ—Add Packed Quadword Integers

Opcode	Instruction	Description
0F D4 /r	PADDQ <i>mm1,mm2/m64</i>	Add quadword integer <i>mm2/m64</i> to <i>mm1</i>
66 0F D4 /r	PADDQ <i>xmm1,xmm2/m128</i>	Add packed quadword integers <i>xmm2/m128</i> to <i>xmm1</i>

### 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 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 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 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.

### Operation

PADDQ instruction with 64-Bit operands:  
 $DEST[63-0] \leftarrow DEST[63-0] + SRC[63-0];$

PADDQ instruction with 128-Bit operands:  
 $DEST[63-0] \leftarrow DEST[63-0] + SRC[63-0];$   
 $DEST[127-64] \leftarrow DEST[127-64] + SRC[127-64];$

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

PADDQ            `__m64 _mm_add_si64 (__m64 a, __m64 b)`

PADDQ            `__m128i _mm_add_epi64 (__m128i a, __m128i b)`

### 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.

(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

**PADDQ—Add Packed Quadword Integers (Continued)**

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

**Numeric Exceptions**

None.

## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode	Instruction	Description
0F EC /r	PADDSB <i>mm, mm/m64</i>	Add packed signed byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F EC /r	PADDSB <i>xmm1,</i>	Add packed signed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> saturate the results.
0F ED /r	PADDSW <i>mm, mm/m64</i>	Add packed signed word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F ED /r	PADDSW <i>xmm1, xmm2/m128</i>	Add packed signed word integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.

### 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 *IA-32 Intel Architecture 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.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX register and the source operand can be either an MMX register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

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.

### Operation

PADDSB instruction 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 instruction 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]);
```

## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation (Continued)

PADDSW instruction 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 instruction 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]);

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

PADDSB        \_\_m64 \_mm\_adds\_pi8(\_\_m64 m1, \_\_m64 m2)

PADDSB        \_\_m128i \_mm\_adds\_epi8 ( \_\_m128i a, \_\_m128i b)

PADDSW        \_\_m64 \_mm\_adds\_pi16(\_\_m64 m1, \_\_m64 m2)

PADDSW        \_\_m128i \_mm\_adds\_epi16 ( \_\_m128i a, \_\_m128i b)

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation (Continued)

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.



## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode	Instruction	Description
0F DC /r	PADDUSB <i>mm, mm/m64</i>	Add packed unsigned byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F DC /r	PADDUSB <i>xmm1, xmm2/m128</i>	Add packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> saturate the results.
0F DD /r	PADDUSW <i>mm, mm/m64</i>	Add packed unsigned word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F DD /r	PADDUSW <i>xmm1, xmm2/m128</i>	Add packed unsigned word integers from <i>xmm2/m128</i> to <i>xmm1</i> and saturate the results.

### 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 *IA-32 Intel Architecture 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. When operating on 64-bit operands, the destination operand must be an MMX register and the source operand can be either an MMX register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PADDUSB instruction adds packed unsigned byte integers. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

The PADDUSW instruction adds packed unsigned word integers. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

### Operation

PADDUSB instruction 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 instruction 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]);
```

## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation (Continued)

PADDUSW instruction 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 instruction 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]);
```

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

```
PADDUSB    __m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW    __m64 _mm_adds_pu16(__m64 m1, __m64 m2)
PADDUSB    __m128i _mm_adds_epu8 ( __m128i a, __m128i b)
PADDUSW    __m128i _mm_adds_epu16 ( __m128i a, __m128i b)
```

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation (Continued)

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PAND—Logical AND

Opcode	Instruction	Description
0F DB /r	PAND <i>mm</i> , <i>mm/m64</i>	Bitwise AND <i>mm/m64</i> and <i>mm</i> .
66 0F DB /r	PAND <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise AND of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical AND operation on 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 MMX register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX register or an XMM register. 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.

### Operation

DEST ← DEST AND SRC;

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

PAND            \_\_m64 \_mm\_and\_si64 ( \_\_m64 m1, \_\_m64 m2)

PAND            \_\_m128i \_mm\_and\_si128 ( \_\_m128i a, \_\_m128i b)

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.

## PAND—Logical AND (Continued)

#AC(0) (64-bit operations only.) 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) (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

(128-bit operations only.) If OSFXSR in CR4 is 0.

(128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PANDN—Logical AND NOT

Opcode	Instruction	Description
0F DF /r	PANDN <i>mm</i> , <i>mm/m64</i>	Bitwise AND NOT of <i>mm/m64</i> and <i>mm</i> .
66 0F DF /r	PANDN <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical NOT of the destination operand (first operand), then performs a bitwise logical AND of the source operand (second operand) and the inverted destination operand. The result is stored in the destination operand. The source operand can be an MMX register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX register or an XMM register. 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.

### Operation

DEST ← (NOT DEST) AND SRC;

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

PANDN        `__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)`

PANDN        `__m128i _mm_andnot_si128 (__m128i a, __m128i b)`

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.

## PANDN—Logical AND NOT (Continued)

#AC(0) (64-bit operations only.) 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) (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

(128-bit operations only.) If OSFXSR in CR4 is 0.

(128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PAUSE—Spin Loop Hint

Opcode	Instruction	Description
F3 90	PAUSE	Gives hint to processor that improves performance of spin-wait loops.

### Description

Improves the performance of spin-wait loops. When executing a “spin-wait loop,” a Pentium 4 processor suffers 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 bypass 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 Pentium 4 processor while executing a spin loop. The Pentium 4 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 processor implements the PAUSE instruction as a pre-defined 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).

### Operation

Execute\_Next\_Instruction(Delay);

### Protected Mode Exceptions

None.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

None.

### Numeric Exceptions

None.



## PAVGB/PAVGW—Average Packed Integers

Opcode	Instruction	Description
0F E0 /r	PAVGB <i>mm1, mm2/m64</i>	Average packed unsigned byte integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 0F E0, /r	PAVGB <i>xmm1, xmm2/m128</i>	Average packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.
0F E3 /r	PAVGW <i>mm1, mm2/m64</i>	Average packed unsigned word integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 0F E3 /r	PAVGW <i>xmm1, xmm2/m128</i>	Average packed unsigned word integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.

### 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 source operand can be an MMX register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX register or an XMM register.

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

### Operation

PAVGB instruction with 64-bit operands:

$$\text{SRC}[7-0] \leftarrow (\text{SRC}[7-0] + \text{DEST}[7-0] + 1) \gg 1; \text{ * temp sum before shifting is 9 bits *}$$

\* repeat operation performed for bytes 2 through 6;

$$\text{SRC}[63-56] \leftarrow (\text{SRC}[63-56] + \text{DEST}[63-56] + 1) \gg 1;$$

PAVGW instruction with 64-bit operands:

$$\text{SRC}[15-0] \leftarrow (\text{SRC}[15-0] + \text{DEST}[15-0] + 1) \gg 1; \text{ * temp sum before shifting is 17 bits *}$$

\* repeat operation performed for words 2 and 3;

$$\text{SRC}[63-48] \leftarrow (\text{SRC}[63-48] + \text{DEST}[63-48] + 1) \gg 1;$$

PAVGB instruction with 128-bit operands:

$$\text{SRC}[7-0] \leftarrow (\text{SRC}[7-0] + \text{DEST}[7-0] + 1) \gg 1; \text{ * temp sum before shifting is 9 bits *}$$

\* repeat operation performed for bytes 2 through 14;

$$\text{SRC}[63-56] \leftarrow (\text{SRC}[63-56] + \text{DEST}[63-56] + 1) \gg 1;$$

PAVGW instruction with 128-bit operands:

$$\text{SRC}[15-0] \leftarrow (\text{SRC}[15-0] + \text{DEST}[15-0] + 1) \gg 1; \text{ * temp sum before shifting is 17 bits *}$$

\* repeat operation performed for words 2 through 6;

$$\text{SRC}[127-48] \leftarrow (\text{SRC}[127-112] + \text{DEST}[127-112] + 1) \gg 1;$$

**PAVGB/PAVGW—Average Packed Integers (Continued)****Intel C/C++ Compiler Intrinsic Equivalent**

PAVGB	<code>__m64_mm_avg_pu8</code> ( <code>__m64 a</code> , <code>__m64 b</code> )
PAVGW	<code>__m64_mm_avg_pu16</code> ( <code>__m64 a</code> , <code>__m64 b</code> )
PAVGB	<code>__m128i_mm_avg_epu8</code> ( <code>__m128i a</code> , <code>__m128i b</code> )
PAVGW	<code>__m128i_mm_avg_epu16</code> ( <code>__m128i a</code> , <code>__m128i b</code> )

**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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.

## PAVGB/PAVGW—Average Packed Integers (Continued)

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode	Instruction	Description
0F 74 /r	PCMPEQB <i>mm</i> , <i>mm/m64</i>	Compare packed bytes in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 74 /r	PCMPEQB <i>xmm1</i> , <i>xmm2/m128</i>	Compare packed bytes in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
0F 75 /r	PCMPEQW <i>mm</i> , <i>mm/m64</i>	Compare packed words in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 75 /r	PCMPEQW <i>xmm1</i> , <i>xmm2/m128</i>	Compare packed words in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
0F 76 /r	PCMPEQD <i>mm</i> , <i>mm/m64</i>	Compare packed doublewords in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 76 /r	PCMPEQD <i>xmm1</i> , <i>xmm2/m128</i>	Compare packed doublewords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.

### 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 source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

The PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the PCMPEQW instruction compares the corresponding words in the destination and source operands; and the PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

### Operation

PCMPEQB instruction with 64-bit operands:

```
IF DEST[7..0] = SRC[7..0]
  THEN DEST[7 0] ← FFH;
  ELSE DEST[7..0] ← 0;
```

\* 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;
```

PCMPEQB instruction with 128-bit operands:

```
IF DEST[7..0] = SRC[7..0]
  THEN DEST[7 0] ← FFH;
  ELSE DEST[7..0] ← 0;
```

## PCMPEQB/PCMPEQW/PCMPEQD—Compare Packed Data for Equal (Continued)

\* Continue comparison of 2nd through 15th bytes in DEST and SRC \*

```
IF DEST[63..56] = SRC[63..56]
    THEN DEST[63..56] ← FFH;
    ELSE DEST[63..56] ← 0;
```

PCMPEQW instruction with 64-bit operands:

```
IF DEST[15..0] = SRC[15..0]
    THEN DEST[15..0] ← FFFFH;
    ELSE DEST[15..0] ← 0;
```

\* 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;
```

PCMPEQW instruction with 128-bit operands:

```
IF DEST[15..0] = SRC[15..0]
    THEN DEST[15..0] ← FFFFH;
    ELSE DEST[15..0] ← 0;
```

\* Continue comparison of 2nd through 7th words in DEST and SRC \*

```
IF DEST[63..48] = SRC[63..48]
    THEN DEST[63..48] ← FFFFH;
    ELSE DEST[63..48] ← 0;
```

PCMPEQD instruction with 64-bit operands:

```
IF DEST[31..0] = SRC[31..0]
    THEN DEST[31..0] ← FFFFFFFFH;
    ELSE DEST[31..0] ← 0;
IF DEST[63..32] = SRC[63..32]
    THEN DEST[63..32] ← FFFFFFFFH;
    ELSE DEST[63..32] ← 0;
```

PCMPEQD instruction with 128-bit operands:

```
IF DEST[31..0] = SRC[31..0]
    THEN DEST[31..0] ← FFFFFFFFH;
    ELSE DEST[31..0] ← 0;
```

\* Continue comparison of 2nd and 3rd doublewords in DEST and SRC \*

```
IF DEST[63..32] = SRC[63..32]
    THEN DEST[63..32] ← FFFFFFFFH;
    ELSE DEST[63..32] ← 0;
```

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

PCMPEQB     \_\_m64 \_mm\_cmpeq\_pi8 (\_\_m64 m1, \_\_m64 m2)

PCMPEQW     \_\_m64 \_mm\_cmpeq\_pi16 (\_\_m64 m1, \_\_m64 m2)

## PCMPEQB/PCMPEQW/PCMPEQD—Compare Packed Data for Equal (Continued)

PCMPEQD	__m64 __mm_cmpeq_pi32 (__m64 m1, __m64 m2)
PCMPEQB	__m128i __mm_cmpeq_epi8 (__m128i a, __m128i b)
PCMPEQW	__m128i __mm_cmpeq_epi16 (__m128i a, __m128i b)
PCMPEQD	__m128i __mm_cmpeq_epi32 (__m128i a, __m128i b)

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.

## PCMPEQB/PCMPEQW/PCMPEQD—Compare Packed Data for Equal (Continued)

- #NM                      If TS in CR0 is set.
- #MF                      (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

- #PF(fault-code)      For a page fault.
- #AC(0)                (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode	Instruction	Description
0F 64 /r	PCMPGTB <i>mm, mm/m64</i>	Compare packed signed byte integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 64 /r	PCMPGTB <i>xmm1, xmm2/m128</i>	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
0F 65 /r	PCMPGTW <i>mm, mm/m64</i>	Compare packed signed word integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 65 /r	PCMPGTW <i>xmm1, xmm2/m128</i>	Compare packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
0F 66 /r	PCMPGTD <i>mm, mm/m64</i>	Compare packed signed doubleword integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 66 /r	PCMPGTD <i>xmm1, xmm2/m128</i>	Compare packed signed doubleword integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.

### Description

Performs a 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 source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

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.

### Operation

PCMPGTB instruction with 64-bit operands:

```
IF DEST[7..0] > SRC[7..0]
  THEN DEST[7..0] ← FFH;
  ELSE DEST[7..0] ← 0;
```

\* 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;
```



## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than (Continued)

PCMPGTB instruction with 128-bit operands:

```
IF DEST[7..0] > SRC[7..0]
  THEN DEST[7..0] ← FFH;
  ELSE DEST[7..0] ← 0;
```

\* Continue comparison of 2nd through 15th bytes in DEST and SRC \*

```
IF DEST[63..56] > SRC[63..56]
  THEN DEST[63..56] ← FFH;
  ELSE DEST[63..56] ← 0;
```

PCMPGTW instruction with 64-bit operands:

```
IF DEST[15..0] > SRC[15..0]
  THEN DEST[15..0] ← FFFFH;
  ELSE DEST[15..0] ← 0;
```

\* 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;
```

PCMPGTW instruction with 128-bit operands:

```
IF DEST[15..0] > SRC[15..0]
  THEN DEST[15..0] ← FFFFH;
  ELSE DEST[15..0] ← 0;
```

\* Continue comparison of 2nd through 7th words in DEST and SRC \*

```
IF DEST[63..48] > SRC[63..48]
  THEN DEST[63..48] ← FFFFH;
  ELSE DEST[63..48] ← 0;
```

PCMPGTD instruction with 64-bit operands:

```
IF DEST[31..0] > SRC[31..0]
  THEN DEST[31..0] ← FFFFFFFFH;
  ELSE DEST[31..0] ← 0;
IF DEST[63..32] > SRC[63..32]
  THEN DEST[63..32] ← FFFFFFFFH;
  ELSE DEST[63..32] ← 0;
```

PCMPGTD instruction with 128-bit operands:

```
IF DEST[31..0] > SRC[31..0]
  THEN DEST[31..0] ← FFFFFFFFH;
  ELSE DEST[31..0] ← 0;
```

\* Continue comparison of 2nd and 3rd doublewords in DEST and SRC \*

```
IF DEST[63..32] > SRC[63..32]
  THEN DEST[63..32] ← FFFFFFFFH;
  ELSE DEST[63..32] ← 0;
```

## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than (Continued)

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

PCMPGTB	<code>__m64 __mm_cmpgt_pi8</code> ( <code>__m64 m1</code> , <code>__m64 m2</code> )
PCMPGTW	<code>__m64 __mm_pcmpgt_pi16</code> ( <code>__m64 m1</code> , <code>__m64 m2</code> )
DCMPGTD	<code>__m64 __mm_pcmpgt_pi32</code> ( <code>__m64 m1</code> , <code>__m64 m2</code> )
PCMPGTB	<code>__m128i __mm_cmpgt_epi8</code> ( <code>__m128i a</code> , <code>__m128i b</code> )
PCMPGTW	<code>__m128i __mm_cmpgt_epi16</code> ( <code>__m128i a</code> , <code>__m128i b</code> )
DCMPGTD	<code>__m128i __mm_cmpgt_epi32</code> ( <code>__m128i a</code> , <code>__m128i b</code> )

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
--------	--

## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than (Continued)

#UD	If EM in CR0 is set. (128-bit operations only.) If OSFXSR in CR4 is 0. (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PEXTRW—Extract Word

Opcode	Instruction	Description
0F C5 /r ib	PEXTRW <i>r32, mm, imm8</i>	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>r32</i> .
66 0F C5 /r ib	PEXTRW <i>r32, xmm, imm8</i>	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to a <i>r32</i> .

### 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 or an XMM register. The destination operand is the low word of a general-purpose register. The count operand is an 8-bit immediate. When specifying a word location in an MMX register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 4 least-significant bits specify the location. The high word of the destination operand is cleared (set to all 0s).

### Operation

PEXTRW instruction with 64-bit source operand:

```
SEL ← COUNT AND 3H;
TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
r32[15-0] ← TEMP[15-0];
r32[31-16] ← 0000H;
```

PEXTRW instruction with 128-bit source operand:

```
SEL ← COUNT AND 7H;
TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
r32[15-0] ← TEMP[15-0];
r32[31-16] ← 0000H;
```

### 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.

### Protected Mode Exceptions

```
#GP(0)      If a memory operand effective address is outside the CS, DS, ES, FS, or
             GS segment limit.
```

**PEXTRW—Extract Word (Continued)**

	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only.) If OSFXSR in CR4 is 0.
	(128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only.) If OSFXSR in CR4 is 0.
	(128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

**Numeric Exceptions**

None.

## PINSRW—Insert Word

Opcode	Instruction	Description
0F C4 /r ib	PINSRW <i>mm</i> , <i>r32/m16</i> , <i>imm8</i>	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 0F C4 /r ib	PINSRW <i>xmm</i> , <i>r32/m16</i> , <i>imm8</i>	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> .

### 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 register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 4 least-significant bits specify the location.

### Operation

PINSRW instruction 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 instruction with 128-bit source operand:

```
SEL ← COUNT AND 7H;
CASE (determine word position) OF
  SEL ← 0:  MASK ← 000000000000000000000000FFFFH;
  SEL ← 1:  MASK ← 000000000000000000000000FFFF0000H;
  SEL ← 2:  MASK ← 000000000000000000000000FFFF00000000H;
  SEL ← 3:  MASK ← 000000000000000000000000FFFF000000000000H;
  SEL ← 4:  MASK ← 000000000000FFFF0000000000000000000H;
  SEL ← 5:  MASK ← 00000000FFFF000000000000000000000H;
  SEL ← 6:  MASK ← 0000FFFF000000000000000000000000H;
  SEL ← 7:  MASK ← FFFF0000000000000000000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);
```

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

PINSRW     \_\_m64 \_\_mm\_insert\_pi16 (\_\_m64 a, int d, int n)

**PINSRW—Insert Word (Continued)**

PINSRW            \_\_m128i \_\_mm\_insert\_epi16 ( \_\_m128i a, int b, int imm)

**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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

## **PINSRW—Insert Word (Continued)**

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### **Numeric Exceptions**

None.



## PMADDWD—Multiply and Add Packed Integers

Opcode	Instruction	Description
0F F5 /r	PMADDWD <i>mm</i> , <i>mm/m64</i>	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 0F F5 /r	PMADDWD <i>xmm1</i> , <i>xmm2/m128</i>	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> .

### 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 3-6 shows this operation when using 64-bit operands.) The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

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

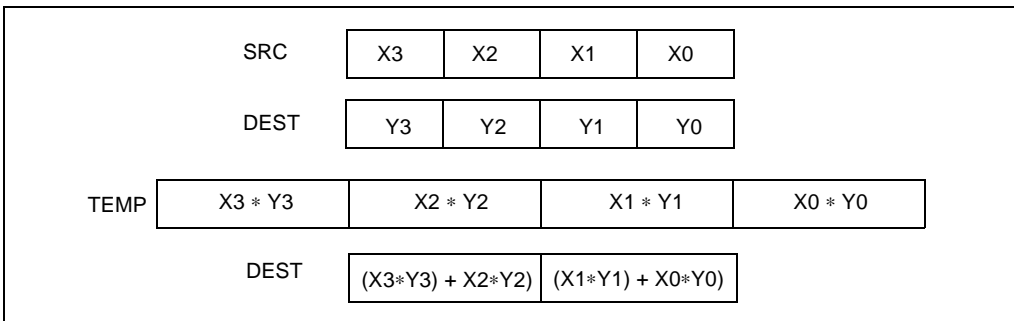


Figure 3-6. PMADDWD Execution Model Using 64-bit Operands

### Operation

PMADDWD instruction with 64-bit operands:

$$\text{DEST}[31..0] \leftarrow (\text{DEST}[15..0] * \text{SRC}[15..0]) + (\text{DEST}[31..16] * \text{SRC}[31..16]);$$

$$\text{DEST}[63..32] \leftarrow (\text{DEST}[47..32] * \text{SRC}[47..32]) + (\text{DEST}[63..48] * \text{SRC}[63..48]);$$

## PMADDWD—Multiply and Add Packed Integers (Continued)

PMADDWD instruction with 128-bit operands:

$$\begin{aligned} \text{DEST}[31..0] &\leftarrow (\text{DEST}[15..0] * \text{SRC}[15..0]) + (\text{DEST}[31..16] * \text{SRC}[31..16]); \\ \text{DEST}[63..32] &\leftarrow (\text{DEST}[47..32] * \text{SRC}[47..32]) + (\text{DEST}[63..48] * \text{SRC}[63..48]); \\ \text{DEST}[95..64] &\leftarrow (\text{DEST}[79..64] * \text{SRC}[79..64]) + (\text{DEST}[95..80] * \text{SRC}[95..80]); \\ \text{DEST}[127..96] &\leftarrow (\text{DEST}[111..96] * \text{SRC}[111..96]) + (\text{DEST}[127..112] * \text{SRC}[127..112]); \end{aligned}$$

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

PMADDWD `__m64 _mm_madd_pi16(__m64 m1, __m64 m2)`

PMADDWD `__m128i _mm_madd_epi16 (__m128i a, __m128i b)`

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
--------	--

**PMADDWD—Multiply and Add Packed Integers (Continued)**

#UD	If EM in CR0 is set. (128-bit operations only.) If OSFXSR in CR4 is 0. (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

**Numeric Exceptions**

None.

## PMAXSW—Maximum of Packed Signed Word Integers

Opcode	Instruction	Description
0F EE /r	PMAXSW <i>mm1, mm2/m64</i>	Compare signed word integers in <i>mm2/m64</i> and <i>mm1</i> and return maximum values.
66 0F EE /r	PMAXSW <i>xmm1, xmm2/m128</i>	Compare signed word integers in <i>xmm2/m128</i> and <i>xmm1</i> and return maximum values.

### 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. The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

### Operation

PMAXSW instruction for 64-bit operands:

```
IF DEST[15-0] > SRC[15-0] THEN
```

```
  (DEST[15-0] ← DEST[15-0];
```

```
ELSE
```

```
  (DEST[15-0] ← SRC[15-0];
```

```
FI
```

\* repeat operation for 2nd and 3rd words in source and destination operands \*

```
IF DEST[63-48] > SRC[63-48] THEN
```

```
  (DEST[63-48] ← DEST[63-48];
```

```
ELSE
```

```
  (DEST[63-48] ← SRC[63-48];
```

```
FI
```

PMAXSW instruction for 128-bit operands:

```
IF DEST[15-0] > SRC[15-0] THEN
```

```
  (DEST[15-0] ← DEST[15-0];
```

```
ELSE
```

```
  (DEST[15-0] ← SRC[15-0];
```

```
FI
```

\* repeat operation for 2nd through 7th words in source and destination operands \*

```
IF DEST[127-112] > SRC[127-112] THEN
```

```
  (DEST[127-112] ← DEST[127-112];
```

```
ELSE
```

```
  (DEST[127-112] ← SRC[127-112];
```

```
FI
```

## PMAXSW—Maximum of Packed Signed Word Integers (Continued)

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

PMAXSW     \_\_m64 \_mm\_max\_pi16(\_\_m64 a, \_\_m64 b)

PMAXSW     \_\_m128i \_mm\_max\_epi16 (\_\_m128i a, \_\_m128i b)

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

**PMAXSW—Maximum of Packed Signed Word Integers (Continued)****Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

**Numeric Exceptions**

None.

## PMAXUB—Maximum of Packed Unsigned Byte Integers

Opcode	Instruction	Description
0F DE /r	PMAXUB <i>mm1, mm2/m64</i>	Compare unsigned byte integers in <i>mm2/m64</i> and <i>mm1</i> and returns maximum values.
66 0F DE /r	PMAXUB <i>xmm1, xmm2/m128</i>	Compare unsigned byte integers in <i>xmm2/m128</i> and <i>xmm1</i> and returns maximum values.

### 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. The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

### Operation

PMAXUB instruction for 64-bit operands:

```
IF DEST[7-0] > SRC[17-0]) THEN
    (DEST[7-0] ← DEST[7-0];
ELSE
    (DEST[7-0] ← SRC[7-0];
FI
```

\* repeat operation for 2nd through 7th bytes in source and destination operands \*

```
IF DEST[63-56] > SRC[63-56]) THEN
    (DEST[63-56] ← DEST[63-56];
ELSE
    (DEST[63-56] ← SRC[63-56];
FI
```

PMAXUB instruction 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];
FI
```

\* repeat operation for 2nd through 15th bytes in source and destination operands \*

```
IF DEST[127-120] > SRC[127-120]) THEN
    (DEST[127-120] ← DEST[127-120];
ELSE
    (DEST[127-120] ← SRC[127-120];
FI
```

## PMAXUB—Maximum of Packed Unsigned Byte Integers (Continued)

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

PMAXUB      `__m64 _mm_max_pu8(__m64 a, __m64 b)`

PMAXUB      `__m128i _mm_max_epu8 (__m128i a, __m128i b)`

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0. (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0. (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.



## **PMAXUB—Maximum of Packed Unsigned Byte Integers (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### **Numeric Exceptions**

None.

## PMINSW—Minimum of Packed Signed Word Integers

Opcode	Instruction	Description
0F EA /r	PMINSW <i>mm1</i> , <i>mm2/m64</i>	Compare signed word integers in <i>mm2/m64</i> and <i>mm1</i> and return minimum values.
66 0F EA /r	PMINSW <i>xmm1</i> , <i>xmm2/m128</i>	Compare signed word integers in <i>xmm2/m128</i> and <i>xmm1</i> and return minimum values.

### 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. The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

### Operation

PMINSW instruction for 64-bit operands:

IF DEST[15-0] < SRC[15-0] THEN

(DEST[15-0] ← DEST[15-0];

ELSE

(DEST[15-0] ← SRC[15-0];

FI

\* repeat operation for 2nd and 3rd words in source and destination operands \*

IF DEST[63-48] < SRC[63-48] THEN

(DEST[63-48] ← DEST[63-48];

ELSE

(DEST[63-48] ← SRC[63-48];

FI

MINSW instruction for 128-bit operands:

IF DEST[15-0] < SRC[15-0] THEN

(DEST[15-0] ← DEST[15-0];

ELSE

(DEST[15-0] ← SRC[15-0];

FI

\* repeat operation for 2nd through 7th words in source and destination operands \*

IF DEST[127-112] < SRC/m64[127-112] THEN

(DEST[127-112] ← DEST[127-112];

ELSE

(DEST[127-112] ← SRC[127-112];

FI

## PMINSW—Minimum of Packed Signed Word Integers (Continued)

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

PMINSW        \_\_m64 \_mm\_min\_pi16 ( \_\_m64 a, \_\_m64 b)

PMINSW        \_\_m128i \_mm\_min\_epi16 ( \_\_m128i a, \_\_m128i b)

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

## PMINSW—Minimum of Packed Signed Word Integers (Continued)

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PMINUB—Minimum of Packed Unsigned Byte Integers

Opcode	Instruction	Description
0F DA /r	PMINUB <i>mm1, mm2/m64</i>	Compare unsigned byte integers in <i>mm2/m64</i> and <i>mm1</i> and returns minimum values.
66 0F DA /r	PMINUB <i>xmm1, xmm2/m128</i>	Compare unsigned byte integers in <i>xmm2/m128</i> and <i>xmm1</i> and returns minimum values.

### 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. The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

### Operation

PMINUB instruction for 64-bit operands:

```
IF DEST[7-0] < SRC[17-0]) THEN
  (DEST[7-0] ← DEST[7-0];
ELSE
  (DEST[7-0] ← SRC[7-0];
FI
```

\* repeat operation for 2nd through 7th bytes in source and destination operands \*

```
IF DEST[63-56] < SRC[63-56]) THEN
  (DEST[63-56] ← DEST[63-56];
ELSE
  (DEST[63-56] ← SRC[63-56];
FI
```

PMINUB instruction 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];
FI
```

\* repeat operation for 2nd through 15th bytes in source and destination operands \*

```
IF DEST[127-120] < SRC[127-120]) THEN
  (DEST[127-120] ← DEST[127-120];
ELSE
  (DEST[127-120] ← SRC[127-120];
FI
```

**PMINUB—Minimum of Packed Unsigned Byte Integers (Continued)****Intel C/C++ Compiler Intrinsic Equivalent**

PMINUB        \_\_m64 \_m\_min\_pu8 (\_\_m64 a, \_\_m64 b)

PMINUB        \_\_m128i \_mm\_min\_epu8 (\_\_m128i a, \_\_m128i b)

**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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

## **PMINUB—Minimum of Packed Unsigned Byte Integers (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### **Numeric Exceptions**

None.

## PMOVMSKB—Move Byte Mask

Opcode	Instruction	Description
0F D7 /r	PMOVMSKB r32, mm	Move a byte mask of mm to r32.
66 0F D7 /r	PMOVMSKB r32, xmm	Move a byte mask of xmm to r32.

### Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand). The source operand is an MMX or an XMM register; the destination operand is a general-purpose register. When operating on 64-bit operands, the byte mask is 8 bits; when operating on 128-bit operands, the byte mask is 16-bits.

### Operation

PMOVMSKB instruction with 64-bit source operand:

r32[0] ← SRC[7];

r32[1] ← SRC[15];

\* repeat operation for bytes 2 through 6;

r32[7] ← SRC[63];

r32[31-8] ← 000000H;

PMOVMSKB instruction with 128-bit source operand:

r32[0] ← SRC[7];

r32[1] ← SRC[15];

\* repeat operation for bytes 2 through 14;

r32[15] ← SRC[127];

r32[31-16] ← 0000H;

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

PMOVMSKB int\_mm\_movemask\_pi8(\_\_m64 a)

PMOVMSKB int\_mm\_movemask\_epi8 (\_\_m128i a)

### Flags Affected

None.

### Protected Mode Exceptions

#UD If EM in CR0 is set.

(128-bit operations only.) If OSFXSR in CR4 is 0.

(128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.



## **PMOVMASKB—Move Byte Mask to General-Purpose Register (Continued)**

#MF (64-bit operations only.) 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

### **Numeric Exceptions**

None.

### **Numeric Exceptions**

None.

## PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode	Instruction	Description
0F E4 /r	PMULHUW <i>mm1, mm2/m64</i>	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 0F E4 /r	PMULHUW <i>xmm1, xmm2/m128</i>	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> .

### 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 3-7 shows this operation when using 64-bit operands.) The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

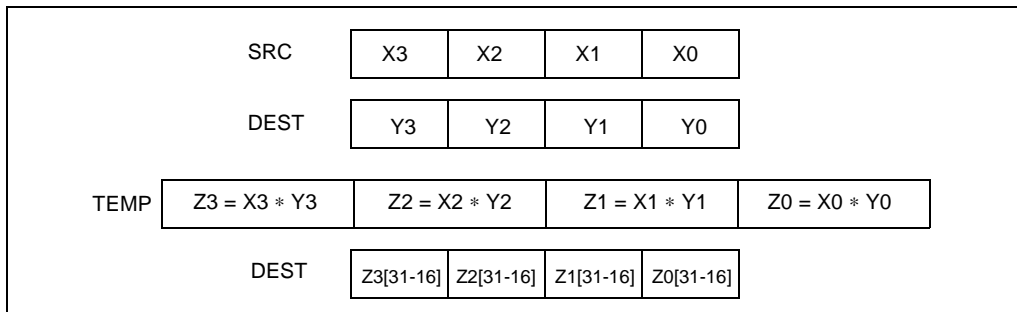


Figure 3-7. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

### Operation

PMULHUW instruction with 64-bit operands:

```

TEMP0[31-0] ← DEST[15-0] * SRC[15-0]; * Unsigned multiplication *
TEMP1[31-0] ← DEST[31-16] * SRC[31-16];
TEMP2[31-0] ← DEST[47-32] * SRC[47-32];
TEMP3[31-0] ← DEST[63-48] * SRC[63-48];
DEST[15-0] ← TEMP0[31-16];
DEST[31-16] ← TEMP1[31-16];
DEST[47-32] ← TEMP2[31-16];
DEST[63-48] ← TEMP3[31-16];

```

## PMULHUW—Multiply Packed Unsigned Integers High (Continued)

PMULHUW instruction with 128-bit operands:

```

TEMP0[31-0] ← DEST[15-0] * SRC[15-0]; * Unsigned multiplication *
TEMP1[31-0] ← DEST[31-16] * SRC[31-16];
TEMP2[31-0] ← DEST[47-32] * SRC[47-32];
TEMP3[31-0] ← DEST[63-48] * SRC[63-48];
TEMP4[31-0] ← DEST[79-64] * SRC[79-64];
TEMP5[31-0] ← DEST[95-80] * SRC[95-80];
TEMP6[31-0] ← DEST[111-96] * SRC[111-96];
TEMP7[31-0] ← DEST[127-112] * SRC[127-112];
DEST[15-0] ← TEMP0[31-16];
DEST[31-16] ← TEMP1[31-16];
DEST[47-32] ← TEMP2[31-16];
DEST[63-48] ← TEMP3[31-16];
DEST[79-64] ← TEMP4[31-16];
DEST[95-80] ← TEMP5[31-16];
DEST[111-96] ← TEMP6[31-16];
DEST[127-112] ← TEMP7[31-16];

```

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

PMULHUW     \_\_m64 \_\_mm\_mulhi\_pu16(\_\_m64 a, \_\_m64 b)

PMULHUW     \_\_m128i \_\_mm\_mulhi\_epu16 (\_\_m128i a, \_\_m128i b)

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.

## PMULHUW—Multiply Packed Unsigned Integers High (Continued)

#AC(0) (64-bit operations only.) 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) (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

(128-bit operations only.) If OSFXSR in CR4 is 0.

(128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode	Instruction	Description
0F E5 /r	PMULHW <i>mm</i> , <i>mm/m64</i>	Multiply the packed signed 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 0F E5 /r	PMULHW <i>xmm1</i> , <i>xmm2/m128</i>	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .

### 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 3-7 shows this operation when using 64-bit operands.) The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

### Operation

PMULHW instruction with 64-bit operands:

```

TEMP0[31-0] ← DEST[15-0] * SRC[15-0]; * Signed multiplication *
TEMP1[31-0] ← DEST[31-16] * SRC[31-16];
TEMP2[31-0] ← DEST[47-32] * SRC[47-32];
TEMP3[31-0] ← DEST[63-48] * SRC[63-48];
DEST[15-0] ← TEMP0[31-16];
DEST[31-16] ← TEMP1[31-16];
DEST[47-32] ← TEMP2[31-16];
DEST[63-48] ← TEMP3[31-16];

```

PMULHW instruction 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];

```

## PMULHW—Multiply Packed Signed Integers and Store High Result (Continued)

DEST[111-96] ← TEMP6[31-16];  
 DEST[127-112] ← TEMP7[31-16];

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

PMULHW     \_\_m64 \_mm\_mulhi\_pi16 (\_\_m64 m1, \_\_m64 m2)  
 PMULHW     \_\_m128i \_mm\_mulhi\_epi16 (\_\_m128i a, \_\_m128i b)

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.

## **PMULHW—Multiply Packed Signed Integers and Store High Result (Continued)**

#NM                      If TS in CR0 is set.

#MF                      (64-bit operations only.) If there is a pending x87 FPU exception.

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

#AC(0)                (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### **Numeric Exceptions**

None.

## PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode	Instruction	Description
0F D5 /r	PMULLW <i>mm</i> , <i>mm/m64</i>	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 0F D5 /r	PMULLW <i>xmm1</i> , <i>xmm2/m128</i>	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> .

### 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 3-7 shows this operation when using 64-bit operands.) The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register.

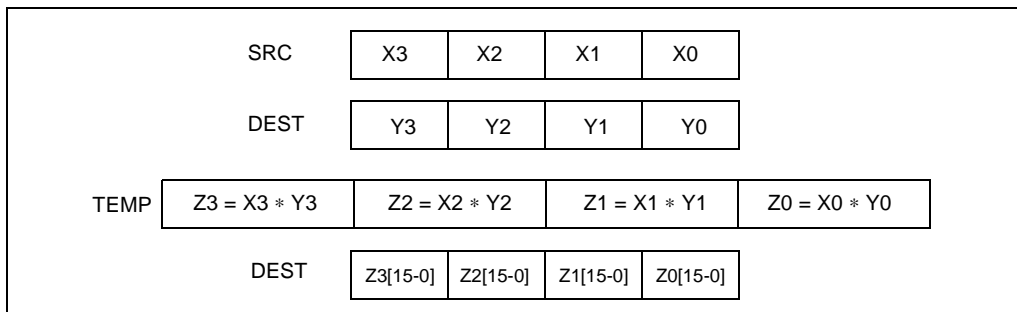


Figure 3-8. PMULLU Instruction Operation Using 64-bit Operands

### Operation

PMULLW instruction with 64-bit operands:

```

TEMP0[31-0] ← DEST[15-0] * SRC[15-0]; * Signed multiplication *
TEMP1[31-0] ← DEST[31-16] * SRC[31-16];
TEMP2[31-0] ← DEST[47-32] * SRC[47-32];
TEMP3[31-0] ← DEST[63-48] * SRC[63-48];
DEST[15-0] ← TEMP0[15-0];
DEST[31-16] ← TEMP1[15-0];
DEST[47-32] ← TEMP2[15-0];
DEST[63-48] ← TEMP3[15-0];

```

PMULLW instruction with 64-bit operands:

```

TEMP0[31-0] ← DEST[15-0] * SRC[15-0]; * Signed multiplication *

```



## PMULLW—Multiply Packed Signed Integers and Store Low Result (Continued)

```

TEMP1[31-0] ← DEST[31-16] * SRC[31-16];
TEMP2[31-0] ← DEST[47-32] * SRC[47-32];
TEMP3[31-0] ← DEST[63-48] * SRC[63-48];
TEMP4[31-0] ← DEST[79-64] * SRC[79-64];
TEMP5[31-0] ← DEST[95-80] * SRC[95-80];
TEMP6[31-0] ← DEST[111-96] * SRC[111-96];
TEMP7[31-0] ← DEST[127-112] * SRC[127-112];
DEST[15-0] ← TEMP0[15-0];
DEST[31-16] ← TEMP1[15-0];
DEST[47-32] ← TEMP2[15-0];
DEST[63-48] ← TEMP3[15-0];
DEST[79-64] ← TEMP4[15-0];
DEST[95-80] ← TEMP5[15-0];
DEST[111-96] ← TEMP6[15-0];
DEST[127-112] ← TEMP7[15-0];

```

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

```

PMULLW    __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
PMULLW    __m128i _mm_mullo_epi16 ( __m128i a, __m128i b)

```

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMULLW—Multiply Packed Signed Integers and Store Low Result (Continued)

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode	Instruction	Description
0F F4 /r	PMULUDQ <i>mm1, mm2/m64</i>	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 <i>xmm1, xmm2/m128</i>	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> .

### Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand. The source operand can be a unsigned doubleword integer stored in the low doubleword of an MMX register or a 64-bit memory location, or it can be two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The destination operand can be a unsigned doubleword integer stored in the low doubleword an MMX register or two packed doubleword integers stored in the first and third doublewords of an XMM register. The result is an unsigned quadword integer stored in the destination an MMX register or two packed unsigned quadword integers stored in an XMM register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation; for 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

### Operation

PMULUDQ instruction with 64-Bit operands:

$$\text{DEST}[63-0] \leftarrow \text{DEST}[31-0] * \text{SRC}[31-0];$$

PMULUDQ instruction with 128-Bit operands:

$$\text{DEST}[63-0] \leftarrow \text{DEST}[31-0] * \text{SRC}[31-0];$$

$$\text{DEST}[127-64] \leftarrow \text{DEST}[95-64] * \text{SRC}[95-64];$$

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

PMULUDQ     \_\_m64 \_mm\_mul\_su32 (\_\_m64 a, \_\_m64 b)

PMULUDQ     \_\_m128i \_mm\_mul\_epu32 (\_\_m128i a, \_\_m128i b)

### Flags Affected

None.

## PMULUDQ—Multiply Packed Unsigned Doubleword Integers (Continued)

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

## POP—Pop a Value from the Stack

Opcode	Instruction	Description
8F /0	POP <i>m16</i>	Pop top of stack into <i>m16</i> ; increment stack pointer
8F /0	POP <i>m32</i>	Pop top of stack into <i>m32</i> ; increment stack pointer
58+ <i>rw</i>	POP <i>r16</i>	Pop top of stack into <i>r16</i> ; increment stack pointer
58+ <i>rd</i>	POP <i>r32</i>	Pop top of stack into <i>r32</i> ; increment stack pointer
1F	POP DS	Pop top of stack into DS; increment stack pointer
07	POP ES	Pop top of stack into ES; increment stack pointer
17	POP SS	Pop top of stack into SS; increment stack pointer
0F A1	POP FS	Pop top of stack into FS; increment stack pointer
0F A9	POP GS	Pop top of stack into GS; increment stack pointer

### Description

Loads the value from the top of the stack to the location specified with the destination operand and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits—the source address size), and the operand-size attribute of the current code segment determines the amount the stack pointer is incremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is incremented by 4 and, if they are 16, the 16-bit SP register is incremented by 2. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the destination operand.)

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 0h as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

## POP—Pop a Value from the Stack (Continued)

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.

### Operation

```

IF StackAddrSize ← 32
  THEN
    IF OperandSize ← 32
      THEN
        DEST ← SS:ESP; (* copy a doubleword *)
        ESP ← ESP + 4;
      ELSE (* OperandSize ← 16*)
        DEST ← SS:ESP; (* copy a word *)
        ESP ← ESP + 2;
    FI;
  ELSE (* StackAddrSize ← 16* )
    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 checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```

IF SS is loaded;
  THEN
    IF segment selector is null
      THEN #GP(0);

```

---

1. Note that in a sequence of instructions that individually delay interrupts past the following instruction, only the first instruction in the sequence is guaranteed to delay the interrupt, but subsequent interrupt-delaying instructions may not delay the interrupt. Thus, in the following instruction sequence:

```

STI
POP SS
POP ESP

```

interrupts may be recognized before the POP ESP executes, because STI also delays interrupts for one instruction.

**POP—Pop a Value from the Stack (Continued)**

```

    FI;
    IF segment selector index is outside descriptor table limits
      OR segment selector's RPL ≠ CPL
      OR segment is not a writable data segment
      OR DPL ≠ CPL
      THEN #GP(selector);
    FI;
    IF segment not marked present
      THEN #SS(selector);
  ELSE
    SS ← segment selector;
    SS ← segment descriptor;
  FI;
FI;
IF DS, ES, FS, or GS is loaded with non-null selector;
THEN
  IF segment selector index is outside descriptor table limits
    OR segment is not a data or readable code segment
    OR ((segment is a data or nonconforming code segment)
      AND (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 DS, ES, FS, or GS is loaded with a null selector;
  THEN
    SegmentRegister ← segment selector;
    SegmentRegister ← segment descriptor;
FI;

```

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If attempt is made to load SS register with null segment selector.
	If the destination operand is in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

**POP—Pop a Value from the Stack (Continued)**

	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.  If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.  If the SS register is being loaded and the segment pointed to is a nonwritable data segment.  If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.  If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the 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.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS 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.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.



## POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Description
61	POPA	Pop DI, SI, BP, BX, DX, CX, and AX
61	POPAD	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX

### 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.)

### Operation

```
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;
```

## POPA/POPAD—Pop All General-Purpose Registers (Continued)

### 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.

### Real-Address Mode Exceptions

#SS	If the starting or ending stack address is not within the stack segment.
-----	--

### 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.

## POPF/POPFD—Pop Stack into EFLAGS Register

Opcode	Instruction	Description
9D	POPF	Pop top of stack into lower 16 bits of EFLAGS
9D	POPFD	Pop top of stack into EFLAGS

### 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 and the POPFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPF is used and to 32 when POPFD is used. Others may treat these mnemonics as synonyms (POPF/POPFD) 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 effect of the POPF/POPFD instructions on the EFLAGS register changes slightly, depending on the mode of operation of the processor. When the processor is operating in protected mode at privilege level 0 (or in real-address mode, which is equivalent to privilege level 0), all the non-reserved flags in the EFLAGS register except the VIP, VIF, and VM flags can be modified. The VIP and VIF flags are cleared, and the VM flag is unaffected.

When operating in protected mode, with a privilege level greater than 0, but less than or equal to IOPL, all the flags can be modified except the IOPL field and the VIP, VIF, and VM flags. 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 the privileged bits do not change.

When operating in virtual-8086 mode, the I/O privilege level (IOPL) must be equal to 3 to use POPF/POPFD instructions and the VM, RF, IOPL, VIP, and VIF flags are unaffected. If the IOPL is less than 3, the POPF/POPFD instructions cause a general-protection exception (#GP).

See the section titled “EFLAGS Register” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for information about the EFLAGS registers.

### Operation

```
IF VM=0 (* Not in Virtual-8086 Mode *)
  THEN IF CPL=0
    THEN
      IF OperandSize ← 32;
        THEN
```

**POPF/POPFD—Pop Stack into EFLAGS Register (Continued)**

```

        EFLAGS ← Pop();
        (* All non-reserved flags except VIP, VIF, and VM can be modified; *)
        (* VIP and VIF are cleared; VM is unaffected*)
    ELSE (* OperandSize ← 16 *)
        EFLAGS[15:0] ← Pop(); (* All non-reserved flags can be modified; *)
    FI;
ELSE (* CPL > 0 *)
    IF OperandSize ← 32;
        THEN
            EFLAGS ← Pop()
            (* All non-reserved bits except IOPL, VIP, and VIF can be modified; *)
            (* IOPL is unaffected; VIP and VIF are cleared; VM is unaffected *)
        ELSE (* OperandSize ← 16 *)
            EFLAGS[15:0] ← Pop();
            (* All non-reserved bits except IOPL can be modified *)
            (* IOPL is 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, and VIF are unaffected *)
            ELSE
                EFLAGS[15:0] ← Pop()
                (* All non-reserved bits except IOPL can be modified *)
                (* IOPL is unaffected *)
            FI;
        ELSE (* IOPL < 3 *)
            #GP(0); (* trap to virtual-8086 monitor *)
        FI;
    FI;
FI;

```

**Flags Affected**

All flags except the reserved bits and the VM bit.

**Protected Mode Exceptions**

- |                 |  |
|-----------------|--|
| #SS(0)          | If the top of stack is not within the stack segment. |
| #PF(fault-code) | If a page fault occurs.                              |

## POPF/POPFD—Pop Stack into EFLAGS Register (Continued)

#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

### Real-Address Mode Exceptions

#SS If the top of stack is not within the stack segment.

### 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.

## POR—Bitwise Logical OR

Opcode	Instruction	Description
0F EB /r	POR <i>mm</i> , <i>mm/m64</i>	Bitwise OR of <i>mm/m64</i> and <i>mm</i> .
66 0F EB /r	POR <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .

### 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. The source operand can be an MMX register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX register or an XMM register. 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.

### Operation

DEST ← DEST OR SRC;

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

POR `__m64 __mm_or_si64(__m64 m1, __m64 m2)`

POR `__m128i __mm_or_si128(__m128i m1, __m128i m2)`

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0. (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.

## POR—Bitwise Logical OR (Continued)

#AC(0) (64-bit operations only.) 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) (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

(128-bit operations only.) If OSFXSR in CR4 is 0.

(128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PREFETCH $h$ —Prefetch Data Into Caches

Opcode	Instruction	Description
0F 18 /1	PREFETCHT0 $m8$	Move data from $m8$ closer to the processor using T0 hint.
0F 18 /2	PREFETCHT1 $m8$	Move data from $m8$ closer to the processor using T1 hint.
0F 18 /3	PREFETCHT2 $m8$	Move data from $m8$ closer to the processor using T2 hint.
0F 18 /0	PREFETCHNTA $m8$	Move data from $m8$ closer to the processor using NTA hint.

### 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 cache levels.
- T1 (temporal data with respect to first level cache)—prefetch data in all cache levels except 0th cache level
- T2 (temporal data with respect to second level cache)—prefetch data in all cache levels, except 0th and 1st cache levels.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure. (This hint can be used to minimize pollution of caches.)

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCH $h$  instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data 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 $h$  instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCH $h$  instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCH $h$  instruction is also unordered with respect to CLFLUSH instructions, other PREFETCH $h$  instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, and OUT, and MOV CR.



## PREFETCH $h$ —Prefetch (Continued)

### 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.

### Protected Mode Exceptions

None.

### Real Address Mode Exceptions

None.

### Virtual 8086 Mode Exceptions

None.

## PSADBW—Compute Sum of Absolute Differences

Opcode	Instruction	Description
0F F6 /r	PSADBW <i>mm1</i> , <i>mm2/m64</i>	Computes the absolute differences of the packed unsigned byte integers from <i>mm2/m64</i> and <i>mm1</i> ; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r	PSADBW <i>xmm1</i> , <i>xmm2/m128</i>	Computes the absolute differences of the packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> ; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.

### Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (first operand) and from the destination operand (second operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. The source operand can be an MMX register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX register or an XMM register. Figure 3-9 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 to 0s.

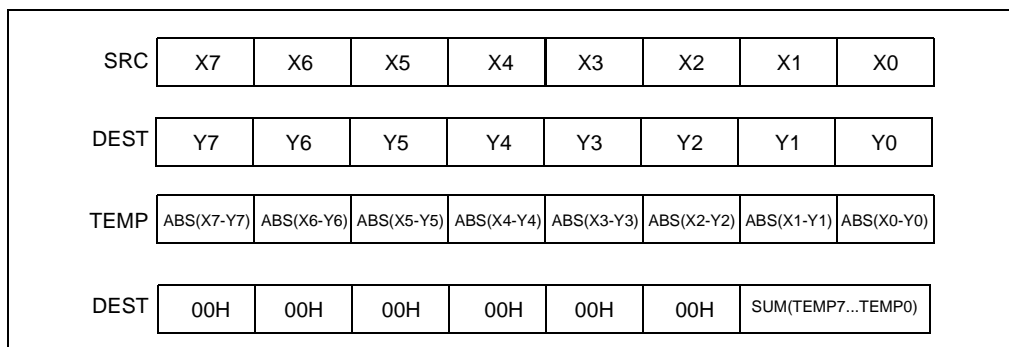


Figure 3-9. PSADBW Instruction Operation Using 64-bit Operands

## PSADBW—Compute Sum of Absolute Differences (Continued)

### Operation

PSADBW instructions when using 64-bit operands:

```

TEMP0 ← ABS(DEST[7-0] – SRC[7-0]);
* repeat operation for bytes 2 through 6 *;
TEMP7 ← ABS(DEST[63-56] – SRC[63-56]);
DEST[15:0] ← SUM(TEMP0...TEMP7);
DEST[63:16] ← 000000000000H;

```

PSADBW instructions when using 128-bit operands:

```

TEMP0 ← ABS(DEST[7-0] – SRC[7-0]);
* repeat operation for bytes 2 through 14 *;
TEMP15 ← ABS(DEST[127-120] – SRC[127-120]);
DEST[15-0] ← SUM(TEMP0...TEMP7);
DEST[63-6] ← 000000000000H;
DEST[79-64] ← SUM(TEMP8...TEMP15);
DEST[127-80] ← 000000000000H;

```

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

PSADBW     \_\_m64\_mm\_sad\_pu8(\_\_m64 a, \_\_m64 b)

PSADBW     \_\_m128i\_mm\_sad\_epu8(\_\_m128i a, \_\_m128i b)

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.

## PSADBW—Compute Sum of Absolute Differences (Continued)

#AC(0) (64-bit operations only.) 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) (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

(128-bit operations only.) If OSFXSR in CR4 is 0.

(128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PSHUFD—Shuffle Packed Doublewords

Opcode	Instruction	Description
66 0F 70 /r ib	PSHUFD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .

### Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at locations selected with the order operand (third operand). Figure 3-10 shows the operation of the PSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location in the destination operand. For example, bits 0 and 1 of the order operand selects the contents of doubleword 0 of the destination operand. The encoding of bits 0 and 1 of the order operand (see the field encoding in Figure 3-10) determines which doubleword from the source operand will be copied to doubleword 0 of the destination operand.

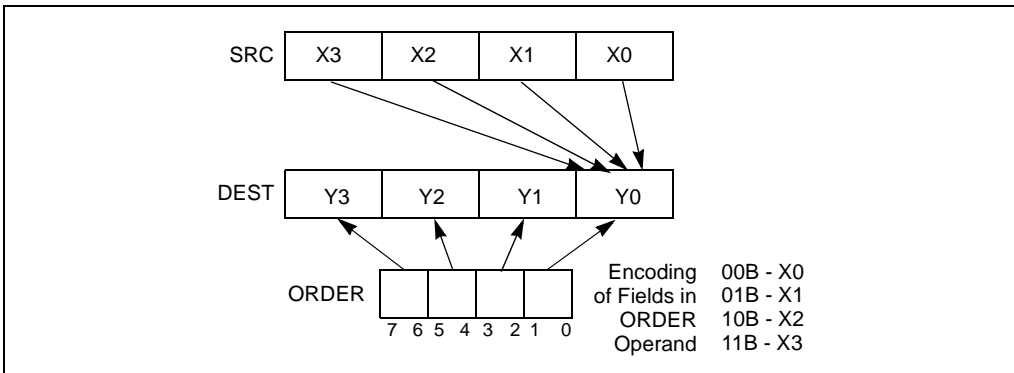


Figure 3-10. PSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate.

Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

### Operation

```

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]
    
```

**PSHUFD—Shuffle Packed Doublewords (Continued)****Intel C/C++ Compiler Intrinsic Equivalent**

PSHUFD      `__m128i _mm_shuffle_epi32(__m128i a, int n)`

**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 memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)      If a memory operand effective address is outside the SS segment limit.

#UD          If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#NM          If TS in CR0 is set.

#PF(fault-code)      If a page fault occurs.

**Real-Address Mode Exceptions**

#GP(0)      If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD          If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#NM          If TS in CR0 is set.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

## **PSHUFD—Shuffle Packed Doublewords (Continued)**

### **Numeric Exceptions**

None.

## PSHUFHW—Shuffle Packed High Words

Opcode	Instruction	Description
F3 0F 70 /r ib	PSHUFHW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	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> .

### Description

Copies words from the high quadword of the source operand (second operand) and inserts them in the high quadword of 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 3-10. For the PSHUFHW instruction, each 2-bit field in the order operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the order 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 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 word in the source operand to be copied to more than one word location in the destination operand.

### Operation

$$\begin{aligned} \text{DEST}[79-64] &\leftarrow (\text{SRC} \gg (\text{ORDER}[1-0] * 16)) [79-64] \\ \text{DEST}[95-80] &\leftarrow (\text{SRC} \gg (\text{ORDER}[3-2] * 16)) [79-64] \\ \text{DEST}[111-96] &\leftarrow (\text{SRC} \gg (\text{ORDER}[5-4] * 16)) [79-64] \\ \text{DEST}[127-112] &\leftarrow (\text{SRC} \gg (\text{ORDER}[7-6] * 16)) [79-64] \end{aligned}$$

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

PSHUFHW     `__m128i _mm_shufflehi_epi16(__m128i a, int n)`

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.



## PSHUFHW—Shuffle Packed High Words (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

### Real-Address Mode Exceptions

#GP(0) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

### Numeric Exceptions

None.

## PSHUFLW—Shuffle Packed Low Words

Opcode	Instruction	Description
F2 0F 70 /r ib	PSHUFLW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	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> .

### Description

Copies words from the low quadword of the source operand (second operand) and inserts them in the low quadword of 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 3-10. For the PSHUFLW instruction, each 2-bit field in the order operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the order 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 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 word in the source operand to be copied to more than one word location in the destination operand.

### Operation

$$\begin{aligned} \text{DEST}[15-0] &\leftarrow (\text{SRC} \gg (\text{ORDER}[1-0] * 16)) [15-0] \\ \text{DEST}[31-16] &\leftarrow (\text{SRC} \gg (\text{ORDER}[3-2] * 16)) [15-0] \\ \text{DEST}[47-32] &\leftarrow (\text{SRC} \gg (\text{ORDER}[5-4] * 16)) [15-0] \\ \text{DEST}[63-48] &\leftarrow (\text{SRC} \gg (\text{ORDER}[7-6] * 16)) [15-0] \end{aligned}$$

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

PSHUFLW     \_\_m128i \_mm\_shufflelo\_epi16(\_\_m128i a, int n)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.

## PSHUFLW—Shuffle Packed Low Words (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

### Real-Address Mode Exceptions

#GP(0) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

### Numeric Exceptions

None.

## PSHUFW—Shuffle Packed Words

Opcode	Instruction	Description
0F 70 /r ib	PSHUFW <i>mm1</i> , <i>mm2/m64</i> , <i>imm8</i>	Shuffle the words in <i>mm2/m64</i> based on the encoding in <i>imm8</i> and store the result in in <i>mm1</i> .

### 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 3-10. 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 register or a 64-bit memory location. The destination operand is an MMX 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.

### Operation

$$\begin{aligned} \text{DEST}[15-0] &\leftarrow (\text{SRC} \gg (\text{ORDER}[1-0] * 16)) [15-0] \\ \text{DEST}[31-16] &\leftarrow (\text{SRC} \gg (\text{ORDER}[3-2] * 16)) [15-0] \\ \text{DEST}[47-32] &\leftarrow (\text{SRC} \gg (\text{ORDER}[5-4] * 16)) [15-0] \\ \text{DEST}[63-48] &\leftarrow (\text{SRC} \gg (\text{ORDER}[7-6] * 16)) [15-0] \end{aligned}$$

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

PSHUFW      `__m64 _mm_shuffle_pi16(__m64 a, int n)`

### 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.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.

## PSHUFW—Shuffle Packed Words (Continued)

#PF(fault-code)	If a 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(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PSLLDQ—Shift Double Quadword Left Logical

Opcode	Instruction	Description
66 0F 73 /7 ib	PSLLDQ <i>xmm1</i> , <i>imm8</i>	Shift <i>xmm1</i> left by <i>imm8</i> bytes while shifting in 0s.

### 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 destination operand is an XMM register. The count operand is an 8-bit immediate.

### Operation

```
TEMP ← COUNT;
if (TEMP > 15) TEMP ← 16;
DEST ← DEST << (TEMP * 8);
```

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

```
PSLLDQ      __m128i _mm_slli_si128 ( __m128i a, int imm)
```

### Flags Affected

None.

### Protected Mode Exceptions

```
#UD          If EM in CR0 is set.
              If OSFXSR in CR4 is 0.
              If CPUID feature flag SSE2 is 0.
#NM          If TS in CR0 is set.
```

### Real-Address Mode Exceptions

Same exceptions as in Protected Mode

### Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode

### Numeric Exceptions

None.

## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode	Instruction	Description
0F F1 /r	PSLLW <i>mm</i> , <i>mm/m64</i>	Shift words in <i>mm</i> left <i>mm/m64</i> while shifting in 0s.
66 0F F1 /r	PSLLW <i>xmm1</i> , <i>xmm2/m128</i>	Shift words in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 71 /6 ib	PSLLW <i>mm</i> , <i>imm8</i>	Shift words in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 71 /6 ib	PSLLW <i>xmm1</i> , <i>imm8</i>	Shift words in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F2 /r	PSLLD <i>mm</i> , <i>mm/m64</i>	Shift doublewords in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F2 /r	PSLLD <i>xmm1</i> , <i>xmm2/m128</i>	Shift doublewords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 72 /6 ib	PSLLD <i>mm</i> , <i>imm8</i>	Shift doublewords in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 72 /6 ib	PSLLD <i>xmm1</i> , <i>imm8</i>	Shift doublewords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F3 /r	PSLLQ <i>mm</i> , <i>mm/m64</i>	Shift quadword in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F3 /r	PSLLQ <i>xmm1</i> , <i>xmm2/m128</i>	Shift quadwords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 73 /6 ib	PSLLQ <i>mm</i> , <i>imm8</i>	Shift quadword in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 73 /6 ib	PSLLQ <i>xmm1</i> , <i>imm8</i>	Shift quadwords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.

### 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 3-11 gives an example of shifting words in a 64-bit operand.) The destination operand may be an MMX register or an XMM register; the count operand can be either an MMX register or an 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate.

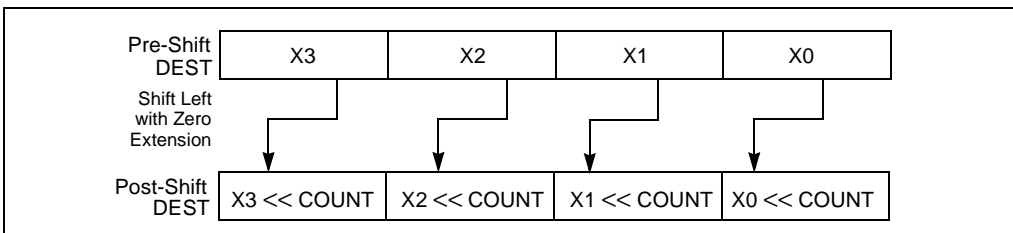


Figure 3-11. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical (Continued)

The 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 PSLLD instruction shifts each of the doublewords in the destination operand; and the PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

### Operation

PSLLW instruction 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 instruction 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 instruction with 64-bit operand:

```
IF (COUNT > 63)
  THEN
    DEST[64..0] ← 0000000000000000H
  ELSE
    DEST ← ZeroExtend(DEST << COUNT);
  FI;
```

PSLLW instruction with 128-bit operand:

```
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;
```



## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical (Continued)

PSLLD instruction with 128-bit operand:

```
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 instruction with 128-bit operand:

```
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;
```

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

PSLLW	__m64 _mm_slli_pi16(__m64 m, int count)
PSLLW	__m64 _mm_sll_pi16(__m64 m, __m64 count)
PSLLW	__m128i _mm_slli_pi16(__m64 m, int count)
PSLLW	__m128i _mm_slli_pi16(__m128i m, __m128i count)
PSLLD	__m64 _mm_slli_pi32(__m64 m, int count)
PSLLD	__m64 _mm_sll_pi32(__m64 m, __m64 count)
PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)
PSLLD	__m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLQ	__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ	__m64 _mm_sll_si64(__m64 m, __m64 count)
PSLLQ	__m128i _mm_slli_si64(__m128i m, int count)
PSLLQ	__m128i _mm_sll_si64(__m128i m, __m128i count)

### Flags Affected

None.

## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical (Continued)

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

## **PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical (Continued)**

### **Numeric Exceptions**

None.

## PSRAW/PSRAD—Shift Packed Data Right Arithmetic

Opcode	Instruction	Description
0F E1 /r	PSRAW <i>mm</i> , <i>mm/m64</i>	Shift words in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 0F E1 /r	PSRAW <i>xmm1</i> , <i>xmm2/m128</i>	Shift words in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
0F 71 /4 ib	PSRAW <i>mm</i> , <i>imm8</i>	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in sign bits
66 0F 71 /4 ib	PSRAW <i>xmm1</i> , <i>imm8</i>	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits
0F E2 /r	PSRAD <i>mm</i> , <i>mm/m64</i>	Shift doublewords in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 0F E2 /r	PSRAD <i>xmm1</i> , <i>xmm2/m128</i>	Shift doubleword in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
0F 72 /4 ib	PSRAD <i>mm</i> , <i>imm8</i>	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 0F 72 /4 ib	PSRAD <i>xmm1</i> , <i>imm8</i>	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.

## 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 3-12 gives an example of shifting words in a 64-bit operand.)

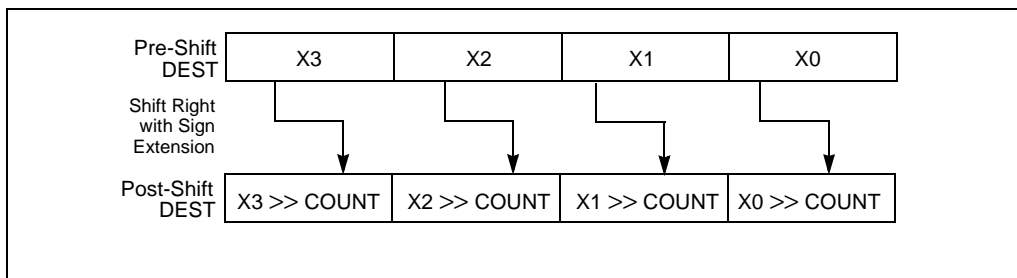


Figure 3-12. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

The destination operand may be an MMX register or an XMM register; the count operand can be either an MMX register or a 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate.

## PSRAW/PSRAD—Shift Packed Data Right Arithmetic (Continued)

The 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 PSRAD instruction shifts each of the doublewords in the destination operand.

### Operation

PSRAW instruction 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 instruction with 64-bit operand:

```
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
ELSE
    DEST[31..0] ← SignExtend(DEST[31..0] >> COUNT);
    DEST[63..32] ← SignExtend(DEST[63..32] >> COUNT);
```

PSRAW instruction with 128-bit operand:

```
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
ELSE
    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 instruction with 128-bit operand:

```
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
ELSE
    DEST[31-0] ← SignExtend(DEST[31-0] >> COUNT);
    * repeat shift operation for 2nd and 3rd doublewords *;
    DEST[127-96] ← SignExtend(DEST[127-96] >>COUNT);
```

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

```
PSRAW    __m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW    __m64 _mm_sraw_pi16 (__m64 m, __m64 count)
PSRAD    __m64 _mm_srai_pi32 (__m64 m, int count)
```

**PSRAW/PSRAD—Shift Packed Data Right Arithmetic (Continued)**

PSRAD	__m64 __mm_sra_pi32 (__m64 m, __m64 count)
PSRAW	__m128i __mm_srai_epi16(__m128i m, int count)
PSRAW	__m128i __mm_sra_epi16(__m128i m, __m128i count)
PSRAD	__m128i __mm_srai_epi32 (__m128i m, int count)
PSRAD	__m128i __mm_sra_epi32 (__m128i m, __m128i count)

**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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.

## PSRAW/PSRAD—Shift Packed Data Right Arithmetic (Continued)

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PSRLDQ—Shift Double Quadword Right Logical

Opcode	Instruction	Description
66 0F 73 /3 ib	PSRLDQ <i>xmm1</i> , <i>imm8</i>	Shift <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.

### 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 destination operand is an XMM register. The count operand is an 8-bit immediate.

### Operation

```
TEMP ← COUNT;
if (TEMP > 15) TEMP ← 16;
DEST ← DEST >> (temp * 8);
```

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

```
PSRLDQ      __m128i _mm_srli_si128 ( __m128i a, int imm)
```

### Flags Affected

None.

### Protected Mode Exceptions

```
#UD          If EM in CR0 is set.
              If OSFXSR in CR4 is 0.
              If CPUID feature flag SSE2 is 0.
#NM          If TS in CR0 is set.
```

### Real-Address Mode Exceptions

Same exceptions as in Protected Mode

### Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode

### Numeric Exceptions

None.



## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode	Instruction	Description
0F D1 /r	PSRLW <i>mm</i> , <i>mm/m64</i>	Shift words in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D1 /r	PSRLW <i>xmm1</i> , <i>xmm2/m128</i>	Shift words in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 71 /2 ib	PSRLW <i>mm</i> , <i>imm8</i>	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 71 /2 ib	PSRLW <i>xmm1</i> , <i>imm8</i>	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
0F D2 /r	PSRLD <i>mm</i> , <i>mm/m64</i>	Shift doublewords in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D2 /r	PSRLD <i>xmm1</i> , <i>xmm2/m128</i>	Shift doublewords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 72 /2 ib	PSRLD <i>mm</i> , <i>imm8</i>	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 72 /2 ib	PSRLD <i>xmm1</i> , <i>imm8</i>	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
0F D3 /r	PSRLQ <i>mm</i> , <i>mm/m64</i>	Shift <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D3 /r	PSRLQ <i>xmm1</i> , <i>xmm2/m128</i>	Shift quadwords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 73 /2 ib	PSRLQ <i>mm</i> , <i>imm8</i>	Shift <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 73 /2 ib	PSRLQ <i>xmm1</i> , <i>imm8</i>	Shift quadwords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.

### 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 3-13 gives an example of shifting words in a 64-bit operand.) The destination operand may be an MMX register or an XMM register; the count operand can be either an MMX register or an 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate.

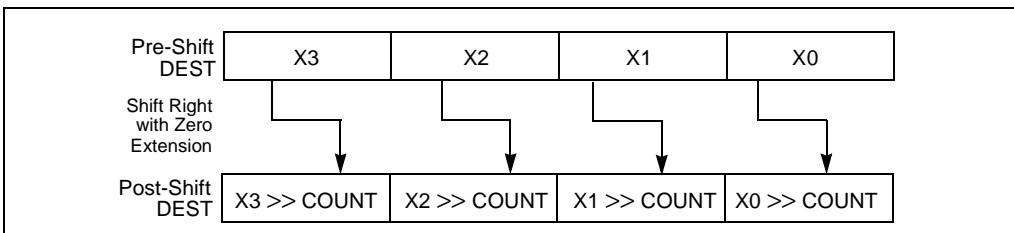


Figure 3-13. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical (Continued)

The 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 PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

### Operation

PSRLW instruction 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 instruction 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 instruction with 64-bit operand:

```
IF (COUNT > 63)
  THEN
    DEST[64..0] ← 0000000000000000H
  ELSE
    DEST ← ZeroExtend(DEST >> COUNT);
  FI;
```

PSRLW instruction with 128-bit operand:

```
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;
```

## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical (Continued)

PSRLD instruction with 128-bit operand:

```
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 instruction with 128-bit operand:

```
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;
```

### 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>
PSRLW	<code>__m128i _mm_srli_epi16 (__m128i m, int count)</code>
PSRLW	<code>__m128i _mm_srl_epi16 (__m128i m, __m128i count)</code>
PSRLD	<code>__m64 _mm_srli_pi32 (__m64 m, int count)</code>
PSRLD	<code>__m64 _mm_srl_pi32 (__m64 m, __m64 count)</code>
PSRLD	<code>__m128i _mm_srli_epi32 (__m128i m, int count)</code>
PSRLD	<code>__m128i _mm_srl_epi32 (__m128i 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>
PSRLQ	<code>__m128i _mm_srli_epi64 (__m128i m, int count)</code>
PSRLQ	<code>__m128i _mm_srl_epi64 (__m128i m, __m128i count)</code>

### Flags Affected

None.

## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical (Continued)

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

## **PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical (Continued)**

### **Numeric Exceptions**

None.

## PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode	Instruction	Description
0F F8 /r	PSUBB <i>mm, mm/m64</i>	Subtract packed byte integers in <i>mm/m64</i> from packed byte integers in <i>mm</i> .
66 0F F8 /r	PSUBB <i>xmm1, xmm2/m128</i>	Subtract packed byte integers in <i>xmm2/m128</i> from packed byte integers in <i>xmm1</i> .
0F F9 /r	PSUBW <i>mm, mm/m64</i>	Subtract packed word integers in <i>mm/m64</i> from packed word integers in <i>mm</i> .
66 0F F9 /r	PSUBW <i>xmm1, xmm2/m128</i>	Subtract packed word integers in <i>xmm2/m128</i> from packed word integers in <i>xmm1</i> .
0F FA /r	PSUBD <i>mm, mm/m64</i>	Subtract packed doubleword integers in <i>mm/m64</i> from packed doubleword integers in <i>mm</i> .
66 0F FA /r	PSUBD <i>xmm1, xmm2/m128</i>	Subtract packed doubleword integers in <i>xmm2/mem128</i> from packed doubleword integers in <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX register and the source operand can be either an MMX register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The 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 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 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 PSUBB, PSUBW, and 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 operated on.

## PSUBB/PSUBW/PSUBD—Subtract Packed Integers (Continued)

### Operation

PSUBB instruction with 64-bit operands:

$DEST[7..0] \leftarrow DEST[7..0] - SRC[7..0];$

\* repeat subtract operation for 2nd through 7th byte \*;

$DEST[63..56] \leftarrow DEST[63..56] - SRC[63..56];$

PSUBB instruction with 128-bit operands:

$DEST[7-0] \leftarrow DEST[7-0] - SRC[7-0];$

\* repeat subtract operation for 2nd through 14th byte \*;

$DEST[127-120] \leftarrow DEST[111-120] - SRC[127-120];$

PSUBW instruction 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 instruction with 128-bit operands:

$DEST[15-0] \leftarrow DEST[15-0] - SRC[15-0];$

\* repeat subtract operation for 2nd through 7th word \*;

$DEST[127-112] \leftarrow DEST[127-112] - SRC[127-112];$

PSUBD instruction with 64-bit operands:

$DEST[31..0] \leftarrow DEST[31..0] - SRC[31..0];$

$DEST[63..32] \leftarrow DEST[63..32] - SRC[63..32];$

PSUBD instruction with 128-bit operands:

$DEST[31-0] \leftarrow DEST[31-0] - SRC[31-0];$

\* repeat subtract operation for 2nd and 3rd doubleword \*;

$DEST[127-96] \leftarrow DEST[127-96] - SRC[127-96];$

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

PSUBB `__m64 _mm_sub_pi8(__m64 m1, __m64 m2)`

PSUBW `__m64 _mm_sub_pi16(__m64 m1, __m64 m2)`

PSUBD `__m64 _mm_sub_pi32(__m64 m1, __m64 m2)`

PSUBB `__m128i _mm_sub_epi8 (__m128i a, __m128i b)`

PSUBW `__m128i _mm_sub_epi16 (__m128i a, __m128i b)`

PSUBD `__m128i _mm_sub_epi32 (__m128i a, __m128i b)`

### Flags Affected

None.

**PSUBB/PSUBW/PSUBD—Subtract Packed Integers (Continued)****Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.



## **PSUBB/PSUBW/PSUBD—Subtract Packed Integers (Continued)**

### **Numeric Exceptions**

None.

## PSUBQ—Subtract Packed Quadword Integers

Opcode	Instruction	Description
0F FB /r	PSUBQ <i>mm1</i> , <i>mm2/m64</i>	Subtract quadword integer in <i>mm1</i> from <i>mm2/m64</i> .
66 0F FB /r	PSUBQ <i>xmm1</i> , <i>xmm2/m128</i>	Subtract packed quadword integers in <i>xmm1</i> from <i>xmm2/m128</i> .

### Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The source operand can be a quadword integer stored in an MMX 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 register or two packed quadword integers stored in an XMM register. 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 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 operated on.

### Operation

PSUBQ instruction with 64-Bit operands:

$$\text{DEST}[63-0] \leftarrow \text{DEST}[63-0] - \text{SRC}[63-0];$$

PSUBQ instruction with 128-Bit operands:

$$\text{DEST}[63-0] \leftarrow \text{DEST}[63-0] - \text{SRC}[63-0];$$

$$\text{DEST}[127-64] \leftarrow \text{DEST}[127-64] - \text{SRC}[127-64];$$

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

PSUBQ            `__m64 _mm_sub_si64(__m64 m1, __m64 m2)`

PSUBQ            `__m128i _mm_sub_epi64(__m128i m1, __m128i m2)`

### 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.

(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

**PSUBQ—Subtract Packed Quadword Integers (Continued)**

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

**Numeric Exceptions**

None.

## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode	Instruction	Description
0F E8 /r	PSUBSB <i>mm, mm/m64</i>	Subtract signed packed bytes in <i>mm/m64</i> from signed packed bytes in <i>mm</i> and saturate results.
66 0F E8 /r	PSUBSB <i>xmm1, xmm2/m128</i>	Subtract packed signed byte integers in <i>xmm2/m128</i> from packed signed byte integers in <i>xmm1</i> and saturate results.
0F E9 /r	PSUBSW <i>mm, mm/m64</i>	Subtract signed packed words in <i>mm/m64</i> from signed packed words in <i>mm</i> and saturate results.
66 0F E9 /r	PSUBSW <i>xmm1, xmm2/m128</i>	Subtract packed signed word integers in <i>xmm2/m128</i> from packed signed word integers in <i>xmm1</i> and saturate results.

### 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 *IA-32 Intel Architecture 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.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX register and the source operand can be either an MMX register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

### Operation

PSUBSB instruction 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/PSUBSW—Subtract Packed Signed Integers with Signed Saturation (Continued)

PSUBSB instruction 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[111-120] – SRC[127-120]);

PSUBSW instruction 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 instruction 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]);

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

PSUBSB        \_\_m64 \_mm\_subs\_pi8(\_\_m64 m1, \_\_m64 m2)  
 PSUBSB        \_\_m128i \_mm\_subs\_epi8(\_\_m128i m1, \_\_m128i m2)  
 PSUBSW        \_\_m64 \_mm\_subs\_pi16(\_\_m64 m1, \_\_m64 m2)  
 PSUBSW        \_\_m128i \_mm\_subs\_epi16(\_\_m128i m1, \_\_m128i m2)

### 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.  
 (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)        If a memory operand effective address is outside the SS segment limit.

#UD            If EM in CR0 is set.  
 (128-bit operations only.) If OSFXSR in CR4 is 0.  
 (128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM            If TS in CR0 is set.

#MF            (64-bit operations only.) If there is a pending x87 FPU exception.

#PF(fault-code)    If a page fault occurs.

## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation (Continued)

#AC(0) (64-bit operations only.) 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) (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

(128-bit operations only.) If OSFXSR in CR4 is 0.

(128-bit operations only.) If CPUID feature flag SSE2 is 0.

#NM If TS in CR0 is set.

#MF (64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode	Instruction	Description
0F D8 /r	PSUBUSB <i>mm</i> , <i>mm/m64</i>	Subtract unsigned packed bytes in <i>mm/m64</i> from unsigned packed bytes in <i>mm</i> and saturate result.
66 0F D8 /r	PSUBUSB <i>xmm1</i> , <i>xmm2/m128</i>	Subtract packed unsigned byte integers in <i>xmm2/m128</i> from packed unsigned byte integers in <i>xmm1</i> and saturate result.
0F D9 /r	PSUBUSW <i>mm</i> , <i>mm/m64</i>	Subtract unsigned packed words in <i>mm/m64</i> from unsigned packed words in <i>mm</i> and saturate result.
66 0F D9 /r	PSUBUSW <i>xmm1</i> , <i>xmm2/m128</i>	Subtract packed unsigned word integers in <i>xmm2/m128</i> from packed unsigned word integers in <i>xmm1</i> and saturate result.

### 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 *IA-32 Intel Architecture 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. When operating on 64-bit operands, the destination operand must be an MMX register and the source operand can be either an MMX register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

### Operation

PSUBUSB instruction with 64-bit operands:

$DEST[7..0] \leftarrow \text{SaturateToUnsignedByte}(DEST[7..0] - SRC[7..0]);$

\* repeat add operation for 2nd through 7th bytes \*

$DEST[63..56] \leftarrow \text{SaturateToUnsignedByte}(DEST[63..56] - SRC[63..56])$

PSUBUSB instruction with 128-bit operands:

$DEST[7-0] \leftarrow \text{SaturateToUnsignedByte}(DEST[7-0] - SRC[7-0]);$

\* repeat add operation for 2nd through 14th bytes \*

$DEST[127-120] \leftarrow \text{SaturateToUnsignedByte}(DEST[127-120] - SRC[127-120]);$

## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation (Continued)

PSUBUSW instruction 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 instruction 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]);
```

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

```
PSUBUSB    __m64 _mm_sub_pu8(__m64 m1, __m64 m2)
PSUBUSB    __m128i _mm_sub_epu8(__m128i m1, __m128i m2)
PSUBUSW    __m64 _mm_sub_pu16(__m64 m1, __m64 m2)
PSUBUSW    __m128i _mm_sub_epu16(__m128i m1, __m128i m2)
```

### 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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation (Continued)

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

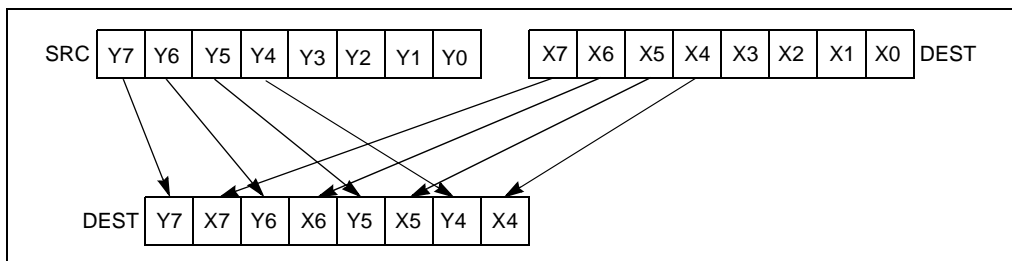
None.

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode	Instruction	Description
0F 68 /r	PUNPCKHBW <i>mm, mm/m64</i>	Unpack and interleave high-order bytes from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 68 /r	PUNPCKHBW <i>xmm1, xmm2/m128</i>	Unpack and interleave high-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 69 /r	PUNPCKHWD <i>mm, mm/m64</i>	Unpack and interleave high-order words from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 69 /r	PUNPCKHWD <i>xmm1, xmm2/m128</i>	Unpack and interleave high-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 6A /r	PUNPCKHDQ <i>mm, mm/m64</i>	Unpack and interleave high-order doublewords from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 6A /r	PUNPCKHDQ <i>xmm1, xmm2/m128</i>	Unpack and interleave high-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 0F 6D /r	PUNPCKHQDQ <i>xmm1, xmm2/m128</i>	Unpack and interleave high-order quadwords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .

### 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 3-14 shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.



**Figure 3-14. PUNPCKHBW Instruction Operation Using 64-bit Operands**

The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register. 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, a processor implementation may fetch only the appropriate 64 bits from memory. Alignment to 16-byte boundary and normal segment checking will still be enforced.

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data (Continued)

The PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

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 PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

### 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];
```

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data (Continued)

```

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];

```

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

```

PUNPCKHBW  __m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
PUNPCKHBW  __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
PUNPCKHWD  __m64 _mm_unpackhi_pi16(__m64 m1, __m64 m2)
PUNPCKHWD  __m128i _mm_unpackhi_epi16(__m128i m1, __m128i m2)
PUNPCKHDQ  __m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
PUNPCKHDQ  __m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
PUNPCKHQDQ __m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)

```

### Flags Affected

None.

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data (Continued)

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#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.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ— Unpack Low Data

Opcode	Instruction	Description
0F 60 /r	PUNPCKLBW <i>mm</i> , <i>mm/m32</i>	Interleave low-order bytes from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 60 /r	PUNPCKLBW <i>xmm1</i> , <i>xmm2/m128</i>	Interleave low-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 61 /r	PUNPCKLWD <i>mm</i> , <i>mm/m32</i>	Interleave low-order words from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 61 /r	PUNPCKLWD <i>xmm1</i> , <i>xmm2/m128</i>	Interleave low-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 62 /r	PUNPCKLDQ <i>mm</i> , <i>mm/m32</i>	Interleave low-order doublewords from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 62 /r	PUNPCKLDQ <i>xmm1</i> , <i>xmm2/m128</i>	Interleave low-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 0F 6C /r	PUNPCKLQDQ <i>xmm1</i> , <i>xmm2/m128</i>	Interleave low-order quadwords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> register

### 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 3-15 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

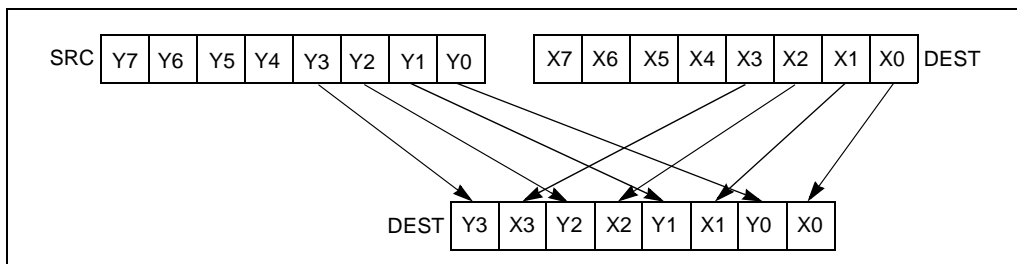


Figure 3-15. PUNPCKLBW Instruction Operation Using 64-bit Operands

The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register. 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, a processor implementation may fetch only the appropriate 64 bits from memory. Alignment to 16-byte boundary and normal segment checking will still be enforced.

## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ— Unpack Low Data (Continued)

The PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

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 PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

### 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];
```

## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ— Unpack Low Data (Continued)

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];

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

PUNPCKLBW `__m64 __mm_unpacklo_pi8 (__m64 m1, __m64 m2)`  
 PUNPCKLBW `__m128i __mm_unpacklo_epi8 (__m128i m1, __m128i m2)`  
 PUNPCKLWD `__m64 __mm_unpacklo_pi16 (__m64 m1, __m64 m2)`  
 PUNPCKLWD `__m128i __mm_unpacklo_epi16 (__m128i m1, __m128i m2)`  
 PUNPCKLDQ `__m64 __mm_unpacklo_pi32 (__m64 m1, __m64 m2)`  
 PUNPCKLDQ `__m128i __mm_unpacklo_epi32 (__m128i m1, __m128i m2)`  
 PUNPCKLQDQ `__m128i __mm_unpacklo_epi64 (__m128i m1, __m128i m2)`

### Flags Affected

None.



## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ— Unpack Low Data (Continued)

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#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.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Numeric Exceptions

None.

## PUSH—Push Word or Doubleword Onto the Stack

Opcode	Instruction	Description
FF /6	PUSH <i>r/m16</i>	Push <i>r/m16</i>
FF /6	PUSH <i>r/m32</i>	Push <i>r/m32</i>
50+ <i>rw</i>	PUSH <i>r16</i>	Push <i>r16</i>
50+ <i>rd</i>	PUSH <i>r32</i>	Push <i>r32</i>
6A	PUSH <i>imm8</i>	Push <i>imm8</i>
68	PUSH <i>imm16</i>	Push <i>imm16</i>
68	PUSH <i>imm32</i>	Push <i>imm32</i>
0E	PUSH CS	Push CS
16	PUSH SS	Push SS
1E	PUSH DS	Push DS
06	PUSH ES	Push ES
0F A0	PUSH FS	Push FS
0F A8	PUSH GS	Push GS

### Description

Decrements the stack pointer and then stores the source operand on the top of the stack. The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits), and the operand-size attribute of the current code segment determines the amount the stack pointer is decremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is decremented by 4 and, if they are 16, the 16-bit SP register is decremented by 2. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the source operand.) Pushing a 16-bit operand when the stack address-size attribute is 32 can result in a misaligned the stack pointer (that is, the stack pointer is not aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus, if a PUSH instruction uses a memory operand in which the ESP register is used as a base register for computing the operand address, the effective address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

### 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 in the real-address and virtual-8086 modes.) 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).

## PUSH—Push Word or Doubleword Onto the Stack (Continued)

### Operation

```

IF StackAddrSize ← 32
THEN
    IF OperandSize ← 32
        THEN
            ESP ← ESP – 4;
            SS:ESP ← SRC; (* push doubleword *)
        ELSE (* OperandSize ← 16*)
            ESP ← ESP – 2;
            SS:ESP ← SRC; (* push word *)
    FI;
ELSE (* StackAddrSize ← 16*)
    IF OperandSize ← 16
        THEN
            SP ← SP – 2;
            SS:SP ← SRC; (* push word *)
        ELSE (* OperandSize ← 32*)
            SP ← SP – 4;
            SS:SP ← SRC; (* push doubleword *)
    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.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
-----	---

**PUSH—Push Word or Doubleword Onto the Stack (Continued)**

- #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.

**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.

## PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Description
60	PUSHA	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

### 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, EBP, 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 the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

### Operation

```

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);
  
```

**PUSHA/PUSHAD—Push All General-Purpose Register (Continued)**

Push(BP);  
Push(SI);  
Push(DI);  
FI;

**Flags Affected**

None.

**Protected Mode Exceptions**

- |                 |  |
|-----------------|--|
| #SS(0)          | If the starting or ending stack address is outside the stack segment limit.  |
| #PF(fault-code) | If a page fault occurs.  |
| #AC(0)          | If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled. |

**Real-Address Mode Exceptions**

- |     |   |
|-----|---|
| #GP | If the ESP or SP register contains 7, 9, 11, 13, or 15. |
|-----|---|

**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. |

## PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

Opcode	Instruction	Description
9C	PUSHF	Push lower 16 bits of EFLAGS
9C	PUSHFD	Push EFLAGS

### 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/POPF 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 the section titled “EFLAGS Register” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for information about the EFLAGS registers.

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.

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, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

### 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;
  FI;
```

## PUSHF/PUSHFD—Push EFLAGS Register onto the Stack (Continued)

ELSE (\* In Virtual-8086 Mode with IOPL less than 0 \*)  
#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.

### Real-Address Mode Exceptions

None.

### 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.



## PXOR—Logical Exclusive OR

Opcode	Instruction	Description
0F EF /r	PXOR <i>mm</i> , <i>mm/m64</i>	Bitwise XOR of <i>mm/m64</i> and <i>mm</i> .
66 0F EF /r	PXOR <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise XOR of <i>xmm2/m128</i> and <i>xmm1</i> .

### 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. The source operand can be an MMX register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX register or an XMM register. 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.

### Operation

DEST ← DEST XOR SRC;

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

PXOR            \_\_m64 \_mm\_xor\_si64 ( \_\_m64 m1, \_\_m64 m2)

PXOR            \_\_m128i \_mm\_xor\_si128 ( \_\_m128i a, \_\_m128i b)

**PXOR—Logical Exclusive OR (Continued)****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.  (128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only.) 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)	(128-bit operations only.) If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.  (128-bit operations only.) If OSFXSR in CR4 is 0.  (128-bit operations only.) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only.) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

**PXOR—Logical Exclusive OR (Continued)**

#AC(0) (64-bit operations only.) If alignment checking is enabled and an unaligned memory reference is made.

**Numeric Exceptions**

None.

## RCL/RCR/ROL/ROR—Rotate

Opcode	Instruction	Description
D0 /2	RCL <i>r/m8</i> , 1	Rotate 9 bits (CF, <i>r/m8</i> ) left once
D2 /2	RCL <i>r/m8</i> , CL	Rotate 9 bits (CF, <i>r/m8</i> ) left CL times
C0 /2 <i>ib</i>	RCL <i>r/m8</i> , <i>imm8</i>	Rotate 9 bits (CF, <i>r/m8</i> ) left <i>imm8</i> times
D1 /2	RCL <i>r/m16</i> , 1	Rotate 17 bits (CF, <i>r/m16</i> ) left once
D3 /2	RCL <i>r/m16</i> , CL	Rotate 17 bits (CF, <i>r/m16</i> ) left CL times
C1 /2 <i>ib</i>	RCL <i>r/m16</i> , <i>imm8</i>	Rotate 17 bits (CF, <i>r/m16</i> ) left <i>imm8</i> times
D1 /2	RCL <i>r/m32</i> , 1	Rotate 33 bits (CF, <i>r/m32</i> ) left once
D3 /2	RCL <i>r/m32</i> , CL	Rotate 33 bits (CF, <i>r/m32</i> ) left CL times
C1 /2 <i>ib</i>	RCL <i>r/m32</i> , <i>imm8</i>	Rotate 33 bits (CF, <i>r/m32</i> ) left <i>imm8</i> times
D0 /3	RCR <i>r/m8</i> , 1	Rotate 9 bits (CF, <i>r/m8</i> ) right once
D2 /3	RCR <i>r/m8</i> , CL	Rotate 9 bits (CF, <i>r/m8</i> ) right CL times
C0 /3 <i>ib</i>	RCR <i>r/m8</i> , <i>imm8</i>	Rotate 9 bits (CF, <i>r/m8</i> ) right <i>imm8</i> times
D1 /3	RCR <i>r/m16</i> , 1	Rotate 17 bits (CF, <i>r/m16</i> ) right once
D3 /3	RCR <i>r/m16</i> , CL	Rotate 17 bits (CF, <i>r/m16</i> ) right CL times
C1 /3 <i>ib</i>	RCR <i>r/m16</i> , <i>imm8</i>	Rotate 17 bits (CF, <i>r/m16</i> ) right <i>imm8</i> times
D1 /3	RCR <i>r/m32</i> , 1	Rotate 33 bits (CF, <i>r/m32</i> ) right once
D3 /3	RCR <i>r/m32</i> , CL	Rotate 33 bits (CF, <i>r/m32</i> ) right CL times
C1 /3 <i>ib</i>	RCR <i>r/m32</i> , <i>imm8</i>	Rotate 33 bits (CF, <i>r/m32</i> ) right <i>imm8</i> times
D0 /0	ROL <i>r/m8</i> , 1	Rotate 8 bits <i>r/m8</i> left once
D2 /0	ROL <i>r/m8</i> , CL	Rotate 8 bits <i>r/m8</i> left CL times
C0 /0 <i>ib</i>	ROL <i>r/m8</i> , <i>imm8</i>	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times
D1 /0	ROL <i>r/m16</i> , 1	Rotate 16 bits <i>r/m16</i> left once
D3 /0	ROL <i>r/m16</i> , CL	Rotate 16 bits <i>r/m16</i> left CL times
C1 /0 <i>ib</i>	ROL <i>r/m16</i> , <i>imm8</i>	Rotate 16 bits <i>r/m16</i> left <i>imm8</i> times
D1 /0	ROL <i>r/m32</i> , 1	Rotate 32 bits <i>r/m32</i> left once
D3 /0	ROL <i>r/m32</i> , CL	Rotate 32 bits <i>r/m32</i> left CL times
C1 /0 <i>ib</i>	ROL <i>r/m32</i> , <i>imm8</i>	Rotate 32 bits <i>r/m32</i> left <i>imm8</i> times
D0 /1	ROR <i>r/m8</i> , 1	Rotate 8 bits <i>r/m8</i> right once
D2 /1	ROR <i>r/m8</i> , CL	Rotate 8 bits <i>r/m8</i> right CL times
C0 /1 <i>ib</i>	ROR <i>r/m8</i> , <i>imm8</i>	Rotate 8 bits <i>r/m8</i> right <i>imm8</i> times
D1 /1	ROR <i>r/m16</i> , 1	Rotate 16 bits <i>r/m16</i> right once
D3 /1	ROR <i>r/m16</i> , CL	Rotate 16 bits <i>r/m16</i> right CL times
C1 /1 <i>ib</i>	ROR <i>r/m16</i> , <i>imm8</i>	Rotate 16 bits <i>r/m16</i> right <i>imm8</i> times
D1 /1	ROR <i>r/m32</i> , 1	Rotate 32 bits <i>r/m32</i> right once
D3 /1	ROR <i>r/m32</i> , CL	Rotate 32 bits <i>r/m32</i> right CL times
C1 /1 <i>ib</i>	ROR <i>r/m32</i> , <i>imm8</i>	Rotate 32 bits <i>r/m32</i> right <i>imm8</i> times

## RCL/RCR/ROL/ROR—Rotate (Continued)

### Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The 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 (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). 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 (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

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 (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). 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 that 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.

### 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;

ESAC;

**RCL/RCR/ROL/ROR—Rotate (Continued)**

(\* 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 \*)

SIZE ← OperandSize

CASE (determine count) OF

```
  SIZE ← 8:   tempCOUNT ← COUNT MOD 8;
  SIZE ← 16:  tempCOUNT ← COUNT MOD 16;
  SIZE ← 32:  tempCOUNT ← COUNT MOD 32;
```

ESAC;

(\* ROL instruction operation \*)

```
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← MSB(DEST);
    DEST ← (DEST * 2) + tempCF;
    tempCOUNT ← tempCOUNT - 1;
  OD;
ELIHW;
CF ← LSB(DEST);
IF COUNT ← 1
  THEN OF ← MSB(DEST) XOR CF;
  ELSE OF is undefined;
```

FI;

**RCL/RCR/ROL/ROR—Rotate (Continued)**

```
(* ROR instruction operation *)
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← LSB(SRC);
    DEST ← (DEST / 2) + (tempCF * 2SIZE);
    tempCOUNT ← tempCOUNT - 1;
  OD;
ELIHW;
CF ← MSB(DEST);
IF COUNT ← 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 nonwritable 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**

#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.

## RCL/RCR/ROL/ROR—Rotate (Continued)

#PF(fault-code)      If a page fault occurs.

#AC(0)                If alignment checking is enabled and an unaligned memory reference is made.



## RCPSS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
OF 53 /r	RCPSS <i>xmm1</i> , <i>xmm2/m128</i>	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> .

### 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 maximum relative error for this approximation is ( $\leq 1.5 * 2^{-12}$ ). 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD single-precision floating-point operation.

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Underflow results are always flushed to 0.0, with the sign of the operand. When a source value is an SNaN or QNaN, the SNaN converted to a QNaN or the source QNaN is returned.

### Operation

```
DEST[31-0] ← APPROXIMATE(1.0/(SRC[31-0]));
DEST[63-32] ← APPROXIMATE(1.0/(SRC[63-32]));
DEST[95-64] ← APPROXIMATE(1.0/(SRC[95-64]));
DEST[127-96] ← APPROXIMATE(1.0/(SRC[127-96]));
```

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

```
RCCPS    __m128 _mm_rcp_ps(__m128 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.
- If memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #SS(0) For an illegal address in the SS segment.

## RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values (Continued)

#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 53 /r	RCPSS <i>xmm1</i> , <i>xmm2/m32</i>	Computes the approximate reciprocal of the scalar single-precision floating-point value in <i>xmm2/m128</i> and stores the result in <i>xmm1</i> .

### Description

Computes of an approximate reciprocal 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 maximum relative error for this approximation is  $(\leq 1.5 * 2^{-12})$ . 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Underflow results are always flushed to 0.0, with the sign of the operand. When a source value is an SNaN or QNaN, the SNaN converted to a QNaN or the source QNaN is returned.

### Operation

DEST[31-0] ← APPROX (1.0/(SRC[31-0]));  
 \* DEST[127-32] remains unchanged \*;

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

RCPSS            \_\_m128 \_mm\_rcp\_ss(\_\_m128 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.
#NM	If TS in CR0 is set.

## RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values (Continued)

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made. For unaligned memory reference if the current privilege level is 3.

## RDMSR—Read from Model Specific Register

Opcode	Instruction	Description
0F 32	RDMSR	Load MSR specified by ECX into EDX:EAX

### Description

Loads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. The input value loaded into the ECX register is the address of the MSR to be read. 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. If less 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. Appendix B, *Model-Specific Registers (MSRs)*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, 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 (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.

### 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.

### Real-Address Mode Exceptions

#GP                        If the value in ECX specifies a reserved or unimplemented MSR address.

## RDMSR—Read from Model Specific Register (Continued)

### Virtual-8086 Mode Exceptions

#GP(0)                    The RDMSR instruction is not recognized in virtual-8086 mode.

## RDPMC—Read Performance-Monitoring Counters

Opcode	Instruction	Description
0F 33	RDPMC	Read performance-monitoring counter specified by ECX into EDX:EAX

### Description

Loads the contents of the 40-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 8 bits of the counter and the EAX register is loaded with the low-order 32 bits. The counter to be read is specified with an unsigned integer placed in the ECX register. The P6 processors have two performance-monitoring counters (0 and 1), which are specified by placing 0000H or 0001H, respectively, in the ECX register. The Pentium 4 processors have 18 counters (0 through 17), which are specified with 0000H through 0011H, respectively.

The Pentium 4 processors also support “fast” (32-bit) and “slow” (40-bit) reads of the performance counters, selected with bit 31 of the ECX register. If bit 31 is set, the RDPMC instruction reads only the low 32 bits of the selected performance counter; if bit 31 is clear, all 40 bits of the counter are read. The 32-bit counter result is returned in the EAX register, and the EDX register is set to 0. A 32-bit read executes faster on a Pentium 4 processor than a full 40-bit read.

The RDPMC instruction allows application code running at a privilege level of 1, 2, or 3 to read the performance-monitoring counters if the PCE flag in the CR4 register is set. This instruction is provided to allow performance monitoring by application code without incurring the overhead of a call to an operating-system procedure.

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. Appendix A, *Performance-Monitoring Events*, in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, lists the events that can be counted for the Pentium 4 earlier IA-32 processors.

The RDPMC instruction is not a serialize 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 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.

**RDPMC—Read Performance-Monitoring Counters (Continued)****Operation**

(\* P6 and Pentium with MMX processors \*)

IF (ECX = 0 OR 1) AND ((CR4.PCE = 1) OR ((CR4.PCE = 0) AND (CPL = 0)))  
THEN

EAX ← PMC(ECX)[31:0];

EDX ← PMC(ECX)[39:32];

ELSE (\* ECX is not 0 or 1 and/or CR4.PCE is 0 and CPL is 1, 2, or 3 \*)

#GP(0); FI;

(\* Pentium 4 processors \*)

IF (ECX = 0...17) AND ((CR4.PCE = 1) OR ((CR4.PCE = 0) AND (CPL = 0)))

THEN IF ECX[31] = 0

THEN EAX ← PMC(ECX)[31:0]; (\* 40-bit read \*);

EDX ← PMC(ECX)[39:32];

ELSE IF ECX[31] = 1

THEN EAX ← PMC(ECX)[31:0]; (\* 32-bit read \*);

EDX ← 0;

FI;

FI;

ELSE (\* ECX ≠ 0...17 and/or CR4.PCE is 0 and CPL is 1, 2, or 3 \*)

#GP(0); FI;

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.

If the value in the ECX register is not 0 or 1.

**Real-Address Mode Exceptions**

#GP If the PCE flag in the CR4 register is clear.

If the value in the ECX register is not 0 or 1.

**Virtual-8086 Mode Exceptions**

#GP(0) If the PCE flag in the CR4 register is clear.

If the value in the ECX register is not 0 or 1.



## RDTSC—Read Time-Stamp Counter

Opcode	Instruction	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX

### Description

Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction. 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. 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. Thus, 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.

This instruction was introduced into the IA-32 Architecture in the Pentium processor.

### Operation

```
IF (CR4.TSD ← 0) OR ((CR4.TSD ← 1) AND (CPL=0))
  THEN
    EDX:EAX ← TimeStampCounter;
  ELSE (* CR4 is 1 and CPL is 1, 2, or 3 *)
    #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.

### Real-Address Mode Exceptions

#GP If the TSD flag in register CR4 is set.

### Virtual-8086 Mode Exceptions

#GP(0) If the TSD flag in register CR4 is set.

## REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	Description
F3 6C	REP INS <i>r/m8</i> , DX	Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS <i>r/m16</i> , DX	Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS <i>r/m32</i> , DX	Input (E)CX doublewords from port DX into ES:[(E)DI]
F3 A4	REP MOVS <i>m8</i> , <i>m8</i>	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS <i>m16</i> , <i>m16</i>	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS <i>m32</i> , <i>m32</i>	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI]
F3 6E	REP OUTS DX, <i>r/m8</i>	Output (E)CX bytes from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX, <i>r/m16</i>	Output (E)CX words from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX, <i>r/m32</i>	Output (E)CX doublewords from DS:[(E)SI] to port DX
F3 AC	REP LODS AL	Load (E)CX bytes from DS:[(E)SI] to AL
F3 AD	REP LODS AX	Load (E)CX words from DS:[(E)SI] to AX
F3 AD	REP LODS EAX	Load (E)CX doublewords from DS:[(E)SI] to EAX
F3 AA	REP STOS <i>m8</i>	Fill (E)CX bytes at ES:[(E)DI] with AL
F3 AB	REP STOS <i>m16</i>	Fill (E)CX words at ES:[(E)DI] with AX
F3 AB	REP STOS <i>m32</i>	Fill (E)CX doublewords at ES:[(E)DI] with EAX
F3 A6	REPE CMPS <i>m8</i> , <i>m8</i>	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI]
F3 A7	REPE CMPS <i>m16</i> , <i>m16</i>	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI]
F3 A7	REPE CMPS <i>m32</i> , <i>m32</i>	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI]
F3 AE	REPE SCAS <i>m8</i>	Find non-AL byte starting at ES:[(E)DI]
F3 AF	REPE SCAS <i>m16</i>	Find non-AX word starting at ES:[(E)DI]
F3 AF	REPE SCAS <i>m32</i>	Find non-EAX doubleword starting at ES:[(E)DI]
F2 A6	REPNE CMPS <i>m8</i> , <i>m8</i>	Find matching bytes in ES:[(E)DI] and DS:[(E)SI]
F2 A7	REPNE CMPS <i>m16</i> , <i>m16</i>	Find matching words in ES:[(E)DI] and DS:[(E)SI]
F2 A7	REPNE CMPS <i>m32</i> , <i>m32</i>	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI]
F2 AE	REPNE SCAS <i>m8</i>	Find AL, starting at ES:[(E)DI]
F2 AF	REPNE SCAS <i>m16</i>	Find AX, starting at ES:[(E)DI]
F2 AF	REPNE SCAS <i>m32</i>	Find EAX, starting at ES:[(E)DI]

### Description

Repeats a string instruction the number of times specified in the count register ((E)CX) 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.

## REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (Continued)

All of these repeat prefixes cause the associated instruction to be repeated until the count in register (E)CX is decremented to 0 (see the following table). (If the current address-size attribute is 32, register ECX is used as a counter, and if the address-size attribute is 16, the CX register is used.) 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 (E)CX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	None
REPE/REPZ	ECX=0	ZF=0
REPNE/REPZ	ECX=0	ZF=1

When the REPE/REPZ and REPNE/REPZ 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.

A REP STOS instruction is the fastest way to initialize a large block of memory.

## REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (Continued)

### Operation

```

IF AddressSize ← 16
  THEN
    use CX for CountReg;
  ELSE (* AddressSize ← 32 *)
    use ECX for CountReg;
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)

None; however, exceptions can be generated by the instruction a repeat prefix is associated with.

## RET—Return from Procedure

Opcode	Instruction	Description
C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2 <i>iw</i>	RET <i>imm16</i>	Near return to calling procedure and pop <i>imm16</i> bytes from stack
CA <i>iw</i>	RET <i>imm16</i>	Far return to calling procedure and pop <i>imm16</i> bytes from stack

### 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 *IA-32 Intel Architecture 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.

## RET—Return from Procedure (Continued)

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).

### Operation

(\* Near return \*)

IF instruction  $\leftarrow$  near return

THEN;

IF OperandSize  $\leftarrow$  32

THEN

IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;

EIP  $\leftarrow$  Pop();

ELSE (\* OperandSize  $\leftarrow$  16 \*)

IF top 6 bytes of stack not within stack limits

THEN #SS(0)

FI;

tempEIP  $\leftarrow$  Pop();

tempEIP  $\leftarrow$  tempEIP AND 0000FFFFH;

IF tempEIP not within code segment limits THEN #GP(0); FI;

EIP  $\leftarrow$  tempEIP;

FI;

IF instruction has immediate operand

THEN IF StackAddressSize=32

THEN

ESP  $\leftarrow$  ESP + SRC; (\* release parameters from stack \*)

ELSE (\* StackAddressSize=16 \*)

SP  $\leftarrow$  SP + SRC; (\* release parameters from stack \*)

FI;

FI;

(\* Real-address mode or virtual-8086 mode \*)

IF ((PE  $\leftarrow$  0) OR (PE  $\leftarrow$  1 AND VM  $\leftarrow$  1)) AND instruction  $\leftarrow$  far return

THEN;

**RET—Return from Procedure (Continued)**

```

    IF OperandSize ← 32
      THEN
        IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
      ELSE (* OperandSize ← 16 *)
        IF top 6 bytes of stack not within stack limits THEN #SS(0); FI;
        tempEIP ← Pop();
        tempEIP ← tempEIP AND 0000FFFFH;
        IF tempEIP not within code segment limits THEN #GP(0); FI;
        EIP ← tempEIP;
        CS ← Pop(); (* 16-bit pop *)
      FI;
  IF instruction has immediate operand
    THEN
      SP ← SP + (SRC AND FFFFH); (* release parameters from stack *)
    FI;
  FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE ← 1 AND VM ← 0) AND instruction ← far RET
  THEN
    IF OperandSize ← 32
      THEN
        IF second doubleword on stack is not within stack limits THEN #SS(0); FI;
      ELSE (* OperandSize ← 16 *)
        IF second word on stack is not within stack limits THEN #SS(0); FI;
      FI;
    IF return code segment selector is null THEN GP(0); FI;
    IF return code segment selector addresss descriptor beyond diescriptor 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 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;
  END;FI;

```

**RET—Return from Procedure (Continued)**

## 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 \*)

ESP ← ESP + SRC; (\* release parameters from stack \*)

ELSE (\* OperandSize=16 \*)

EIP ← Pop();

EIP ← EIP AND 0000FFFFH;

CS ← Pop(); (\* 16-bit pop \*)

ESP ← ESP + SRC; (\* release parameters from stack \*)

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;

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 information also loaded \*)

CS(RPL) ← CPL;

ESP ← ESP + SRC; (\* release parameters from called procedure's stack \*)

tempESP ← Pop();

tempSS ← Pop(); (\* 32-bit pop, high-order 16 bits discarded \*)

(\* segment descriptor information also loaded \*)

ESP ← tempESP;

SS ← tempSS;



**RET—Return from Procedure (Continued)**

```

ELSE (* OperandSize=16 *)
    EIP ← Pop();
    EIP ← EIP AND 0000FFFFH;
    CS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    ESP ← ESP + SRC; (* release parameters from called procedure's stack *)
    tempESP ← Pop();
    tempSS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
    (* segment descriptor information also 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 (* segment register invalid *)
            SegmentSelector ← 0; (* null segment selector *)
        FI;
    OD;
For each of ES, FS, GS, and DS
DO
    IF segment selector index is not within descriptor table limits
    OR segment descriptor indicates the segment is not a data or
    readable code segment
    OR if the segment is a data or non-conforming code segment and the segment
    descriptor's DPL < CPL or RPL of code segment's segment selector
    THEN
        segment selector register ← null selector;
    OD;
ESP ← ESP + SRC; (* release parameters from calling procedure's stack *)

```

**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.

**RET—Return from Procedure (Continued)**

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.

## **ROL/ROR—Rotate**

See entry for RCL/RCR/ROL/ROR—Rotate.

## RSM—Resume from System Management Mode

Opcode	Instruction	Description
0F AA	RSM	Resume operation of interrupted program

### 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 SSM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486 processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

See Chapter 12, *System Management Mode (SMM)*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information about SMM and the behavior of the RSM instruction.

### Operation

```
ReturnFromSSM;
ProcessorState ← Restore(SSMDump);
```

### Flags Affected

All.

### Protected Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.

### Real-Address Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.

### Virtual-8086 Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.

## RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 52 /r	RSQRTPS <i>xmm1</i> , <i>xmm2/m128</i>	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> .

### 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 maximum relative error for this approximation is ( $\leq 1.5 * 2^{-12}$ ). 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD single-precision floating-point operation.

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  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. Underflow results are always flushed to 0.0, with the sign of the operand. When a source value is an SNaN or QNaN, the SNaN converted to a QNaN or the source QNaN is returned.

### Operation

```
DEST[31-0] ← APPROXIMATE(1.0/SQRT(SRC[31-0]));
DEST[63-32] ← APPROXIMATE(1.0/SQRT(SRC[63-32]));
DEST[95-64] ← APPROXIMATE(1.0/SQRT(SRC[95-64]));
DEST[127-96] ← APPROXIMATE(1.0/SQRT(SRC[127-96]));
```

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

```
RSQRTPS    __m128 _mm_rsqrtps(__m128 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.

## RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values (Continued)

	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode	Instruction	Description
F3 0F 52 /r	RSQRTSS <i>xmm1</i> , <i>xmm2/m32</i>	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .

### 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 maximum relative error for this approximation is ( $\leq 1.5 * 2^{-12}$ ). 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  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. Underflow results are always flushed to 0.0, with the sign of the operand. When a source value is an SNaN or QNaN, the SNaN converted to a QNaN or the source QNaN is returned.

### Operation

DEST[31-0] ← APPROXIMATE(1.0/SQRT(SRC[31-0]));  
\* DEST[127-32] remains unchanged \*;

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

RSQRTSS     \_\_m128 \_mm\_rsqr\_tss(\_\_m128 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.

## RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value (Continued)

#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.



## SAHF—Store AH into Flags

Opcode	Instruction	Clocks	Description
9E	SAHF	2	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register

### 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.

### Operation

EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;

### 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.

### Exceptions (All Operating Modes)

None.

## SAL/SAR/SHL/SHR—Shift

Opcode	Instruction	Description
D0 /4	SAL <i>r/m8</i> ,1	Multiply <i>r/m8</i> by 2, once
D2 /4	SAL <i>r/m8</i> ,CL	Multiply <i>r/m8</i> by 2, CL times
C0 /4 <i>ib</i>	SAL <i>r/m8</i> , <i>imm8</i>	Multiply <i>r/m8</i> by 2, <i>imm8</i> times
D1 /4	SAL <i>r/m16</i> ,1	Multiply <i>r/m16</i> by 2, once
D3 /4	SAL <i>r/m16</i> ,CL	Multiply <i>r/m16</i> by 2, CL times
C1 /4 <i>ib</i>	SAL <i>r/m16</i> , <i>imm8</i>	Multiply <i>r/m16</i> by 2, <i>imm8</i> times
D1 /4	SAL <i>r/m32</i> ,1	Multiply <i>r/m32</i> by 2, once
D3 /4	SAL <i>r/m32</i> ,CL	Multiply <i>r/m32</i> by 2, CL times
C1 /4 <i>ib</i>	SAL <i>r/m32</i> , <i>imm8</i>	Multiply <i>r/m32</i> by 2, <i>imm8</i> times
D0 /7	SAR <i>r/m8</i> ,1	Signed divide* <i>r/m8</i> by 2, once
D2 /7	SAR <i>r/m8</i> ,CL	Signed divide* <i>r/m8</i> by 2, CL times
C0 /7 <i>ib</i>	SAR <i>r/m8</i> , <i>imm8</i>	Signed divide* <i>r/m8</i> by 2, <i>imm8</i> times
D1 /7	SAR <i>r/m16</i> ,1	Signed divide* <i>r/m16</i> by 2, once
D3 /7	SAR <i>r/m16</i> ,CL	Signed divide* <i>r/m16</i> by 2, CL times
C1 /7 <i>ib</i>	SAR <i>r/m16</i> , <i>imm8</i>	Signed divide* <i>r/m16</i> by 2, <i>imm8</i> times
D1 /7	SAR <i>r/m32</i> ,1	Signed divide* <i>r/m32</i> by 2, once
D3 /7	SAR <i>r/m32</i> ,CL	Signed divide* <i>r/m32</i> by 2, CL times
C1 /7 <i>ib</i>	SAR <i>r/m32</i> , <i>imm8</i>	Signed divide* <i>r/m32</i> by 2, <i>imm8</i> times
D0 /4	SHL <i>r/m8</i> ,1	Multiply <i>r/m8</i> by 2, once
D2 /4	SHL <i>r/m8</i> ,CL	Multiply <i>r/m8</i> by 2, CL times
C0 /4 <i>ib</i>	SHL <i>r/m8</i> , <i>imm8</i>	Multiply <i>r/m8</i> by 2, <i>imm8</i> times
D1 /4	SHL <i>r/m16</i> ,1	Multiply <i>r/m16</i> by 2, once
D3 /4	SHL <i>r/m16</i> ,CL	Multiply <i>r/m16</i> by 2, CL times
C1 /4 <i>ib</i>	SHL <i>r/m16</i> , <i>imm8</i>	Multiply <i>r/m16</i> by 2, <i>imm8</i> times
D1 /4	SHL <i>r/m32</i> ,1	Multiply <i>r/m32</i> by 2, once
D3 /4	SHL <i>r/m32</i> ,CL	Multiply <i>r/m32</i> by 2, CL times
C1 /4 <i>ib</i>	SHL <i>r/m32</i> , <i>imm8</i>	Multiply <i>r/m32</i> by 2, <i>imm8</i> times
D0 /5	SHR <i>r/m8</i> ,1	Unsigned divide <i>r/m8</i> by 2, once
D2 /5	SHR <i>r/m8</i> ,CL	Unsigned divide <i>r/m8</i> by 2, CL times
C0 /5 <i>ib</i>	SHR <i>r/m8</i> , <i>imm8</i>	Unsigned divide <i>r/m8</i> by 2, <i>imm8</i> times
D1 /5	SHR <i>r/m16</i> ,1	Unsigned divide <i>r/m16</i> by 2, once
D3 /5	SHR <i>r/m16</i> ,CL	Unsigned divide <i>r/m16</i> by 2, CL times
C1 /5 <i>ib</i>	SHR <i>r/m16</i> , <i>imm8</i>	Unsigned divide <i>r/m16</i> by 2, <i>imm8</i> times
D1 /5	SHR <i>r/m32</i> ,1	Unsigned divide <i>r/m32</i> by 2, once
D3 /5	SHR <i>r/m32</i> ,CL	Unsigned divide <i>r/m32</i> by 2, CL times
C1 /5 <i>ib</i>	SHR <i>r/m32</i> , <i>imm8</i>	Unsigned divide <i>r/m32</i> by 2, <i>imm8</i> times

**NOTE:**

\* Not the same form of division as IDIV; rounding is toward negative infinity.

## SAL/SAR/SHL/SHR—Shift (Continued)

### 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 register CL. The count is masked to 5 bits, which limits the count range to 0 to 31. 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 *IA-32 Intel Architecture 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 *IA-32 Intel Architecture 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 *IA-32 Intel Architecture 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 cleared 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.

## SAL/SAR/SHL/SHR—Shift (Continued)

### 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

```
tempCOUNT ← (COUNT AND 1FH);
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 *);
    FI;
  FI;
  tempCOUNT ← tempCOUNT – 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT ← 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);
    FI;
  FI;
FI;
```

**SAL/SAR/SHL/SHR—Shift (Continued)**

```

ELSE IF COUNT ← 0
  THEN
    All flags remain unchanged;
  ELSE (* COUNT neither 1 or 0 *)
    OF ← undefined;

```

```

FI;

```

```

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 nonwritable 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**

#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.

## SBB—Integer Subtraction with Borrow

Opcode	Instruction	Description
1C <i>ib</i>	SBB AL, <i>imm8</i>	Subtract with borrow <i>imm8</i> from AL
1D <i>iw</i>	SBB AX, <i>imm16</i>	Subtract with borrow <i>imm16</i> from AX
1D <i>id</i>	SBB EAX, <i>imm32</i>	Subtract with borrow <i>imm32</i> from EAX
80 /3 <i>ib</i>	SBB <i>r/m8,imm8</i>	Subtract with borrow <i>imm8</i> from <i>r/m8</i>
81 /3 <i>iw</i>	SBB <i>r/m16,imm16</i>	Subtract with borrow <i>imm16</i> from <i>r/m16</i>
81 /3 <i>id</i>	SBB <i>r/m32,imm32</i>	Subtract with borrow <i>imm32</i> from <i>r/m32</i>
83 /3 <i>ib</i>	SBB <i>r/m16,imm8</i>	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m16</i>
83 /3 <i>ib</i>	SBB <i>r/m32,imm8</i>	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m32</i>
18 /r	SBB <i>r/m8,r8</i>	Subtract with borrow <i>r8</i> from <i>r/m8</i>
19 /r	SBB <i>r/m16,r16</i>	Subtract with borrow <i>r16</i> from <i>r/m16</i>
19 /r	SBB <i>r/m32,r32</i>	Subtract with borrow <i>r32</i> from <i>r/m32</i>
1A /r	SBB <i>r8,r/m8</i>	Subtract with borrow <i>r/m8</i> from <i>r8</i>
1B /r	SBB <i>r16,r/m16</i>	Subtract with borrow <i>r/m16</i> from <i>r16</i>
1B /r	SBB <i>r32,r/m32</i>	Subtract with borrow <i>r/m32</i> from <i>r32</i>

### 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.

### Operation

DEST ← DEST – (SRC + CF);

## SBB—Integer Subtraction with Borrow (Continued)

### 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 nonwritable 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

#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.

## SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	Description
AE	SCAS m8	Compare AL with byte at ES:(E)DI and set status flags
AF	SCAS m16	Compare AX with word at ES:(E)DI and set status flags
AF	SCAS m32	Compare EAX with doubleword at ES:(E)DI and set status flags
AE	SCASB	Compare AL with byte at ES:(E)DI and set status flags
AF	SCASW	Compare AX with word at ES:(E)DI and set status flags
AF	SCASD	Compare EAX with doubleword at ES:(E)DI and set status flags

### Description

Compares the byte, word, or double word specified with the memory operand with the value in the AL, AX, or EAX register, and sets the status flags in the EFLAGS register according to the results. The memory operand address is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot 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-operand form (specified with the SCAS mnemonic) allows the memory operand to be specified explicitly. Here, the memory operand should 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 (the AL register for byte comparisons, AX for word comparisons, and EAX for doubleword comparisons). This explicit-operand form is provided to allow documentation; however, 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 the ES:(E)DI 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 SCAS instructions. Here also ES:(E)DI is assumed to be the memory operand and the AL, AX, or EAX register is assumed to be the register operand. The size of the two operands is selected with 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 (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The SCAS, SCASB, SCASW, and SCASD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. 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 this chapter for a description of the REP prefix.



## SCAS/SCASB/SCASW/SCASD—Scan String (Continued)

### Operation

```

IF (byte comparison)
  THEN
    temp ← AL – SRC;
    SetStatusFlags(temp);
    THEN IF DF ← 0
      THEN (E)DI ← (E)DI + 1;
      ELSE (E)DI ← (E)DI – 1;
    FI;
  ELSE IF (word comparison)
    THEN
      temp ← AX – SRC;
      SetStatusFlags(temp)
      THEN IF DF ← 0
        THEN (E)DI ← (E)DI + 2;
        ELSE (E)DI ← (E)DI – 2;
      FI;
    ELSE (* doubleword comparison *)
      temp ← EAX – SRC;
      SetStatusFlags(temp)
      THEN IF DF ← 0
        THEN (E)DI ← (E)DI + 4;
        ELSE (E)DI ← (E)DI – 4;
      FI;
  FI;
FI;

```

### Flags Affected

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

### Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the limit of the ES segment.</p> <p>If the ES register contains a null segment selector.</p> <p>If an illegal memory operand effective address in the ES segment is given.</p>
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SCAS/SCASB/SCASW/SCASD—Scan String (Continued)

### 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.               |

## SETcc—Set Byte on Condition

Opcode	Instruction	Description
0F 97	SETA <i>r/m8</i>	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE <i>r/m8</i>	Set byte if above or equal (CF=0)
0F 92	SETB <i>r/m8</i>	Set byte if below (CF=1)
0F 96	SETBE <i>r/m8</i>	Set byte if below or equal (CF=1 or ZF=1)
0F 92	SETC <i>r/m8</i>	Set if carry (CF=1)
0F 94	SETE <i>r/m8</i>	Set byte if equal (ZF=1)
0F 9F	SETG <i>r/m8</i>	Set byte if greater (ZF=0 and SF=OF)
0F 9D	SETGE <i>r/m8</i>	Set byte if greater or equal (SF=OF)
0F 9C	SETL <i>r/m8</i>	Set byte if less (SF<>OF)
0F 9E	SETLE <i>r/m8</i>	Set byte if less or equal (ZF=1 or SF<>OF)
0F 96	SETNA <i>r/m8</i>	Set byte if not above (CF=1 or ZF=1)
0F 92	SETNAE <i>r/m8</i>	Set byte if not above or equal (CF=1)
0F 93	SETNB <i>r/m8</i>	Set byte if not below (CF=0)
0F 97	SETNBE <i>r/m8</i>	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC <i>r/m8</i>	Set byte if not carry (CF=0)
0F 95	SETNE <i>r/m8</i>	Set byte if not equal (ZF=0)
0F 9E	SETNG <i>r/m8</i>	Set byte if not greater (ZF=1 or SF<>OF)
0F 9C	SETNGE <i>r/m8</i>	Set if not greater or equal (SF<>OF)
0F 9D	SETNL <i>r/m8</i>	Set byte if not less (SF=OF)
0F 9F	SETNLE <i>r/m8</i>	Set byte if not less or equal (ZF=0 and SF=OF)
0F 91	SETNO <i>r/m8</i>	Set byte if not overflow (OF=0)
0F 9B	SETNP <i>r/m8</i>	Set byte if not parity (PF=0)
0F 99	SETNS <i>r/m8</i>	Set byte if not sign (SF=0)
0F 95	SETNZ <i>r/m8</i>	Set byte if not zero (ZF=0)
0F 90	SETO <i>r/m8</i>	Set byte if overflow (OF=1)
0F 9A	SETP <i>r/m8</i>	Set byte if parity (PF=1)
0F 9A	SETPE <i>r/m8</i>	Set byte if parity even (PF=1)
0F 9B	SETPO <i>r/m8</i>	Set byte if parity odd (PF=0)
0F 98	SETS <i>r/m8</i>	Set byte if sign (SF=1)
0F 94	SETZ <i>r/m8</i>	Set byte if zero (ZF=1)

### Description

Set 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.

## SETcc—Set Byte on Condition (Continued)

Many of the SETcc 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 0F. These alternate mnemonics are provided to make code more intelligible. Appendix B, *EFLAGS Condition Codes*, in the *IA-32 Intel Architecture 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 SETcc instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

### 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 nonwritable 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.

### 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.

## SFENCE—Store Fence

Opcode	Instruction	Description
OF AE /7	SFENCE	Serializes store operations.

### 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 in program order the SFENCE instruction is globally visible before any store instruction that follows the SFENCE instruction is globally visible. The SFENCE instruction is ordered with respect store instructions, other SFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of insuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

### Operation

Wait\_On\_Following\_Stores\_Until(preceding\_stores\_globally\_visible);

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

void\_mm\_sfence(void)

### Protected Mode Exceptions

None.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

None.

## SGDT/SIDT—Store Global/Interrupt Descriptor Table Register

Opcode	Instruction	Description
0F 01 /0	SGDT <i>m</i>	Store GDTR to <i>m</i>
0F 01 /1	SIDT <i>m</i>	Store IDTR to <i>m</i>

### Description

Stores the contents of the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the lower 2 bytes of the memory location and the 32-bit base address is stored in the upper 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the lower 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

The SGDT and SIDT instructions are only useful in operating-system software; however, they can be used in application programs without causing an exception to be generated.

See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in this chapter for information on loading the GDTR and IDTR.

### IA-32 Architecture Compatibility

The 16-bit forms of the SGDT and SIDT instructions are 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 Pro, Pentium, Intel486, and Intel386 processors fill these bits with 0s.

### Operation

```

IF instruction is IDTR
  THEN
    IF OperandSize ← 16
      THEN
        DEST[0:15] ← IDTR(Limit);
        DEST[16:39] ← IDTR(Base); (* 24 bits of base address loaded; *)
        DEST[40:47] ← 0;
      ELSE (* 32-bit Operand Size *)
        DEST[0:15] ← IDTR(Limit);
        DEST[16:47] ← IDTR(Base); (* full 32-bit base address loaded *)
      FI;
  ELSE (* instruction is SGDT *)
    IF OperandSize ← 16
      THEN
        DEST[0:15] ← GDTR(Limit);
        DEST[16:39] ← GDTR(Base); (* 24 bits of base address loaded; *)
        DEST[40:47] ← 0;

```

## SGDT/SIDT—Store Global/Interrupt Descriptor Table Register (Continued)

ELSE (\* 32-bit Operand Size \*)  
 DEST[0:15] ← GDTR(Limit);  
 DEST[16:47] ← GDTR(Base); (\* full 32-bit base address loaded \*)  
 FI; FI;

### Flags Affected

None.

### Protected Mode Exceptions

#UD	If the destination operand is a register.
#GP(0)	If the destination is located in a nonwritable 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.
#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.
#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.

## **SHL/SHR—Shift Instructions**

See entry for SAL/SAR/SHL/SHR—Shift.



## SHLD—Double Precision Shift Left

Opcode	Instruction	Description
0F A4	SHLD <i>r/m16, r16, imm8</i>	Shift <i>r/m16</i> to left <i>imm8</i> places while shifting bits from <i>r16</i> in from the right
0F A5	SHLD <i>r/m16, r16, CL</i>	Shift <i>r/m16</i> to left CL places while shifting bits from <i>r16</i> in from the right
0F A4	SHLD <i>r/m32, r32, imm8</i>	Shift <i>r/m32</i> to left <i>imm8</i> places while shifting bits from <i>r32</i> in from the right
0F A5	SHLD <i>r/m32, r32, CL</i>	Shift <i>r/m32</i> to left CL places while shifting bits from <i>r32</i> in from the right

### Description

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 an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand 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, the flags are not affected.

The SHLD instruction is useful for multi-precision shifts of 64 bits or more.

### Operation

COUNT ← COUNT MOD 32;

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 DOWNT0 COUNT

DO

Bit(DEST, *i*) ← Bit(DEST, *i* – COUNT);

OD;

**SHLD—Double Precision Shift Left (Continued)**

```

        FOR i ← COUNT – 1 DOWNT0 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 nonwritable 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**

- #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.

## SHRD—Double Precision Shift Right

Opcode	Instruction	Description
0F AC	SHRD <i>r/m16, r16, imm8</i>	Shift <i>r/m16</i> to right <i>imm8</i> places while shifting bits from <i>r16</i> in from the left
0F AD	SHRD <i>r/m16, r16, CL</i>	Shift <i>r/m16</i> to right CL places while shifting bits from <i>r16</i> in from the left
0F AC	SHRD <i>r/m32, r32, mm8</i>	Shift <i>r/m32</i> to right <i>imm8</i> places while shifting bits from <i>r32</i> in from the left
0F AD	SHRD <i>r/m32, r32, CL</i>	Shift <i>r/m32</i> to right CL places while shifting bits from <i>r32</i> in from the left

### Description

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 an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand 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, the flags are not affected.

The SHRD instruction is useful for multiprecision shifts of 64 bits or more.

### Operation

COUNT ← COUNT MOD 32;

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;

**SHRD—Double Precision Shift Right (Continued)**

```

        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 nonwritable 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**

#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.

## SHUFPD—Shuffle Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F C6 /r ib	SHUFPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	Shuffle packed double-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm1/m128</i> to <i>xmm1</i> .

### 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 to the high quadword of the destination operand (see Figure 3-16). The select operand (third operand) determines which values are moved to the destination operand.

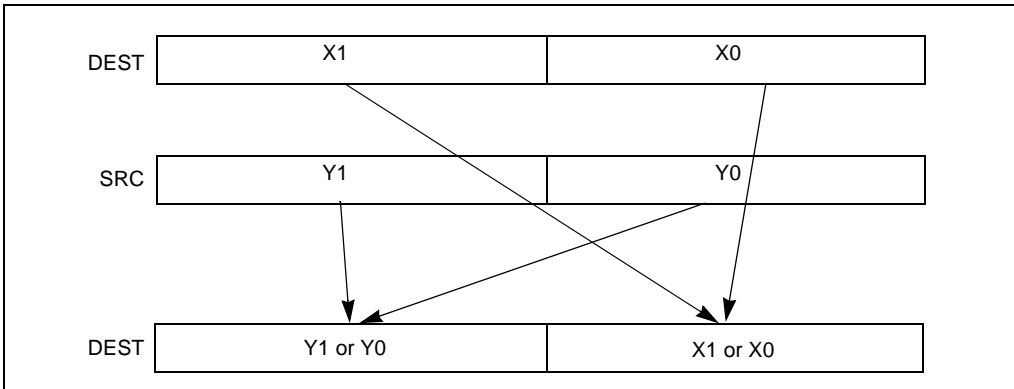


Figure 3-16. SHUFPD Shuffle Operation

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 3 through 7 of the shuffle operand are reserved.

### Operation

```

IF SELECT[0] == 0
    THEN DEST[63-0] ← DEST[63-0];
    ELSE DEST[63-0] ← DEST[127-64]; FI;
IF SELECT[1] == 0
    THEN DEST[127-64] ← SRC[63-0];
    ELSE DEST[127-64] ← SRC[127-64]; FI;
    
```

## SHUFDP—Shuffle Packed Double-Precision Floating-Point Values (Continued)

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

SHUFDP `__m128d __mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)`

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

## SHUFDP—Shuffle Packed Double-Precision Floating-Point Values (Continued)

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

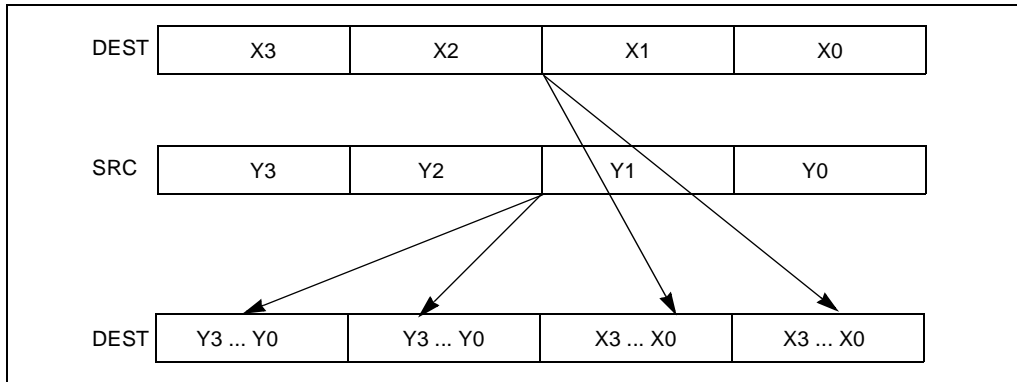
#PF(fault-code) For a page fault.

## SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F C6 /r ib	SHUFPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	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> .

### 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 3-17). The select operand (third operand) determines which values are moved to the destination operand.



**Figure 3-17. 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 the low doubleword of the result, bits 2 and 3 select the value to be moved from the destination operand the second doubleword of the result, bits 4 and 5 select the value to be moved from the source operand the third doubleword of the result, and bits 6 and 7 select the value to be moved from the source operand 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];



## SHUFPS—Shuffle Packed Single-Precision Floating-Point Values (Continued)

```

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;

```

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

SHUFPS `__m128 __mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)`

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

## SHUFPS—Shuffle Packed Single-Precision Floating-Point Values (Continued)

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## **SIDT—Store Interrupt Descriptor Table Register**

See entry for SGDT/SIDT—Store Global/Interrupt Descriptor Table Register.

## SLDT—Store Local Descriptor Table Register

Opcode	Instruction	Description
0F 00 /0	SLDT <i>r/m16</i>	Stores segment selector from LDTR in <i>r/m16</i>
0F 00 /0	SLDT <i>r/m32</i>	Store segment selector from LDTR in low-order 16 bits of <i>r/m32</i>

### 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.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower-order 16 bits of the register. The high-order 16 bits of the register are cleared to 0s for the Pentium Pro processor and 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.

The SLDT instruction is only useful in operating-system software; however, it can be used in application programs.

### Operation

DEST ← LDTR(SegmentSelector);

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable 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.

## SLDT—Store Local Descriptor Table Register (Continued)

### Real-Address Mode Exceptions

#UD                      The SLDT instruction is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD                      The SLDT instruction is not recognized in virtual-8086 mode.

## SMSW—Store Machine Status Word

Opcode	Instruction	Description
0F 01 /4	SMSW <i>r/m16</i>	Store machine status word to <i>r/m16</i>
0F 01 /4	SMSW <i>r32/m16</i>	Store machine status word in low-order 16 bits of <i>r32/m16</i> ; high-order 16 bits of <i>r32</i> are undefined

### Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a 16-bit general-purpose register or a memory location.

When the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the upper 16 bits of the register are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

The SMSW instruction is only useful in operating-system software; however, it is not a privileged instruction and can be used in application programs.

This instruction is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on the Pentium Pro, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the machine status word.

### Operation

DEST ← CR0[15:0]; (\* Machine status word \*);

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable 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.

## SMSW—Store Machine Status Word (Continued)

### 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.

### 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.

## SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 51 /r	SQRTPD <i>xmm1</i> , <i>xmm2/m128</i>	Computes square roots of the packed double-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD double-precision floating-point operation.

### Operation

DEST[63-0] ← SQRT(SRC[63-0]);  
 DEST[127-64] ← SQRT(SRC[127-64]);

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

SQRTPD        `__m128d _mm_sqrt_pd (m128d a)`

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.



## SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 51 /r	SQRTPS <i>xmm1</i> , <i>xmm2/m128</i>	Computes square roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD single-precision floating-point operation.

### Operation

```
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]);
```

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

```
SQRTPS    __m128 _mm_sqrt_ps(__m128 a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

## SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values (Continued)

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
-----	---

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode	Instruction	Description
F2 0F 51 /r	SQRTSD <i>xmm1</i> , <i>xmm2/m64</i>	Computes square root of the low double-precision floating-point value in <i>xmm2/m64</i> and stores the results in <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

### Operation

DEST[63-0] ← SQRT(SRC[63-0]);  
 \* DEST[127-64] remains unchanged \*;

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

SQRTSD        \_\_m128d \_mm\_sqrt\_sd (m128d a)

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0.

## SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value (Continued)

If CPUID feature flag SSE2 is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value

Opcode	Instruction	Description
F3 0F 51 /r	SQRTSS <i>xmm1</i> , <i>xmm2/m32</i>	Computes square root of the low single-precision floating-point value in <i>xmm2/m32</i> and stores the results in <i>xmm1</i> .

### 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 doublewords of the destination operand remains unchanged. See Figure 10-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

### Operation

DEST[31-0] ← SQRT (SRC[31-0]);  
 \* DEST[127-64] remains unchanged \*;

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

SQRTSS      `__m128_mm_sqrt_ss(__m128 a)`

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.

## SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value (Continued)

If CPUID feature flag SSE is 0.

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## STC—Set Carry Flag

Opcode	Instruction	Description
F9	STC	Set CF flag

### Description

Sets the CF flag in the EFLAGS register.

### 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)

None.



## STD—Set Direction Flag

Opcode	Instruction	Description
FD	STD	Set DF flag

### Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI).

### Operation

$DF \leftarrow 1$ ;

### Flags Affected

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

### Operation

$DF \leftarrow 1$ ;

### Exceptions (All Operating Modes)

None.

## STI—Set Interrupt Flag

Opcode	Instruction	Description
FB	STI	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction

### Description

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>. This behavior allows external interrupts to be disabled at the beginning of a procedure and enabled again at the end of the procedure. 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 have no effect on the generation of exceptions and NMI interrupts.

The following decision table indicates the action of the STI instruction (bottom of the table) depending on the processor's mode of operation and the CPL and IOPL of the currently running program or procedure (top of the table).

PE =	0	1	1	1
VM =	X	0	0	1
CPL	X	≤ IOPL	> IOPL	=3
IOPL	X	X	X	=3
IF ← 1	Y	Y	N	Y
#GP(0)	N	N	Y	N

#### NOTES:

X Don't care.

N Action in Column 1 not taken.

Y Action in Column 1 taken.

- Note that in a sequence of instructions that individually delay interrupts past the following instruction, only the first instruction in the sequence is guaranteed to delay the interrupt, but subsequent interrupt-delaying instructions may not delay the interrupt. Thus, in the following instruction sequence:

```
STI
MOV SS, AX
MOV ESP, EBP
```

interrupts may be recognized before MOV ESP, EBP executes, even though MOV SS, AX normally delays interrupts for one instruction.

## STI—Set Interrupt Flag (Continued)

### 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 ← 3
          THEN
            IF ← 1;
          ELSE
            IF CPL ≤ IOPL
              THEN
                IF ← 1;
              ELSE
                #GP(0);
            FI;
          FI;
        ELSE (* Executing in Virtual-8086 mode *)
          #GP(0); (* Trap to virtual-8086 monitor *)
        FI;
      FI;
    FI;
  FI;

```

### Flags Affected

The IF 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.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

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

## STMXCSR—Store MXCSR Register State

Opcode	Instruction	Description
0F AE /3	STMXCSR <i>m32</i>	Store contents of MXCSR register to <i>m32</i> .

### Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

### Operation

$m32 \leftarrow \text{MXCSR};$

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

`_mm_getcsr(void)`

### Exceptions

None.

### Numeric 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 CR0.EM = 1.
#NM	If TS bit in CR0 is set.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).
#UD	If CR4.OSFXSR(bit 9) = 0.
#UD	If CPUID.XMM(EDX bit 25) = 0.

## STMXCSR—Store MXCSR Register State (Continued)

### Real Address Mode Exceptions

Interrupt 13	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#UD	If CR0.EM = 1.
#NM	If TS bit in CR0 is set.
#UD	If CR4.OSFXSR(bit 9) = 0.
#UD	If CPUID.XMM(EDX bit 25) = 0.

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.

## STOS/STOSB/STOSW/STOSD—Store String

Opcode	Instruction	Description
AA	STOS m8	Store AL at address ES:(E)DI
AB	STOS m16	Store AX at address ES:(E)DI
AB	STOS m32	Store EAX at address ES:(E)DI
AA	STOSB	Store AL at address ES:(E)DI
AB	STOSW	Store AX at address ES:(E)DI
AB	STOSD	Store EAX at address ES:(E)DI

### Description

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 the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot 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 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, 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 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 store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and the AL, AX, or EAX register is assumed to be the source operand. The size of the destination and source operands is selected with the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), or STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the AL, AX, or EAX register to the memory location, 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 (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

## STOS/STOSB/STOSW/STOSD—Store String (Continued)

The STOS, STOSB, STOSW, and STOSD 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

```

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;
    ELSE (* doubleword store *)
      DEST ← EAX;
      THEN IF DF ← 0
        THEN (E)DI ← (E)DI + 4;
        ELSE (E)DI ← (E)DI - 4;
      FI;
  FI;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

- |                 |  |
|-----------------|--|
| #GP(0)          | If the destination is located in a nonwritable segment.<br><br>If a memory operand effective address is outside the limit of the ES segment.<br><br>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.   |

**STOS/STOSB/STOSW/STOSD—Store String (Continued)****Real-Address Mode Exceptions**

#GP                      If a memory operand effective address is outside the ES segment limit.

**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.



## STR—Store Task Register

Opcode	Instruction	Description
OF 00 /1	STR <i>r/m16</i>	Stores segment selector from TR in <i>r/m16</i>

### 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 to 0s. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

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 nonwritable 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.

### Real-Address Mode Exceptions

- #UD The STR instruction is not recognized in real-address mode.

## STR—Store Task Register (Continued)

### Virtual-8086 Mode Exceptions

#UD                      The STR instruction is not recognized in virtual-8086 mode.

## SUB—Subtract

Opcode	Instruction	Description
2C <i>ib</i>	SUB AL, <i>imm8</i>	Subtract <i>imm8</i> from AL
2D <i>iw</i>	SUB AX, <i>imm16</i>	Subtract <i>imm16</i> from AX
2D <i>id</i>	SUB EAX, <i>imm32</i>	Subtract <i>imm32</i> from EAX
80 /5 <i>ib</i>	SUB <i>r/m8</i> , <i>imm8</i>	Subtract <i>imm8</i> from <i>r/m8</i>
81 /5 <i>iw</i>	SUB <i>r/m16</i> , <i>imm16</i>	Subtract <i>imm16</i> from <i>r/m16</i>
81 /5 <i>id</i>	SUB <i>r/m32</i> , <i>imm32</i>	Subtract <i>imm32</i> from <i>r/m32</i>
83 /5 <i>ib</i>	SUB <i>r/m16</i> , <i>imm8</i>	Subtract sign-extended <i>imm8</i> from <i>r/m16</i>
83 /5 <i>ib</i>	SUB <i>r/m32</i> , <i>imm8</i>	Subtract sign-extended <i>imm8</i> from <i>r/m32</i>
28 /r	SUB <i>r/m8</i> , <i>r8</i>	Subtract <i>r8</i> from <i>r/m8</i>
29 /r	SUB <i>r/m16</i> , <i>r16</i>	Subtract <i>r16</i> from <i>r/m16</i>
29 /r	SUB <i>r/m32</i> , <i>r32</i>	Subtract <i>r32</i> from <i>r/m32</i>
2A /r	SUB <i>r8</i> , <i>r/m8</i>	Subtract <i>r/m8</i> from <i>r8</i>
2B /r	SUB <i>r16</i> , <i>r/m16</i>	Subtract <i>r/m16</i> from <i>r16</i>
2B /r	SUB <i>r32</i> , <i>r/m32</i>	Subtract <i>r/m32</i> from <i>r32</i>

### 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 a borrow 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.

### Operation

DEST ← DEST – SRC;

### Flags Affected

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

## SUB—Subtract (Continued)

### Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable 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

#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.

## SUBPD—Subtract Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 5C /r	SUBPD <i>xmm1</i> , <i>xmm2/m128</i>	Subtract packed double-precision floating-point values in <i>xmm2/m128</i> from <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD double-precision floating-point operation.

### Operation

DEST[63-0] ← DEST[63-0] – SRC[63-0];  
 DEST[127-64] ← DEST[127-64] – SRC[127-64];

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

SUBPD            \_\_m128d \_mm\_sub\_pd (m128d a, m128d b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.

## SUBPD—Subtract Packed Double-Precision Floating-Point Values (Continued)

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 5C /r	SUBPS <i>xmm1 xmm2/m128</i>	Subtract packed single-precision floating-point values in <i>xmm2/mem</i> from <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD double-precision floating-point operation.

### Operation

```
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];
```

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

```
SUBPS    __m128 _mm_sub_ps(__m128 a, __m128 b)
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

## SUBPS—Subtract Packed Single-Precision Floating-Point Values (Continued)

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------



## SUBSD—Subtract Scalar Double-Precision Floating-Point Values

Opcode	Instruction	Description
F2 0F 5C /r	SUBSD <i>xmm1</i> , <i>xmm2/m64</i>	Subtracts the low double-precision floating-point values in <i>xmm2/mem64</i> from <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

### Operation

DEST[63-0] ← DEST[63-0] – SRC[63-0];

\* DEST[127-64] remains unchanged \*;

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

SUBSD            \_\_m128d \_mm\_sub\_sd (m128d a, m128d b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

## SUBSD—Subtract Scalar Double-Precision Floating-Point Values (Continued)

#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

Interrupt 13            If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM                      If TS in CR0 is set.

#XM                      If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD                      If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)        For a page fault.

#AC(0)                    If alignment checking is enabled and an unaligned memory reference is made.

## SUBSS—Subtract Scalar Single-Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 5C /r	SUBSS <i>xmm1</i> , <i>xmm2/m32</i>	Subtract the lower single-precision floating-point values in <i>xmm2/m32</i> from <i>xmm1</i> .

### 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

### Operation

DEST[31-0] ← DEST[31-0] - SRC[31-0];  
 \* DEST[127-96] remains unchanged \*;

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

SUBSS            \_\_m128 \_mm\_sub\_ss(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

## SUBSS—Subtract Scalar Single-Precision Floating-Point Values (Continued)

#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

Interrupt 13 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If TS in CR0 is set.

#XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## SYSENTER—Fast System Call

Opcode	Instruction	Description
0F 34	SYSENTER	Fast call to privilege level 0 system procedures

### Description

Executes a fast call to a level 0 system procedure or routine. This instruction is a companion instruction to the SYSEXIT instruction. The SYSENTER 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.

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 into the following MSRs:

- **SYSENTER\_CS\_MSR**—Contains the 32-bit segment selector for the privilege level 0 code segment. (This value is also used to compute the segment selector of the privilege level 0 stack segment.)
- **SYSENTER\_EIP\_MSR**—Contains the 32-bit offset into the privilege level 0 code segment to the first instruction of the selected operating procedure or routine.
- **SYSENTER\_ESP\_MSR**—Contains the 32-bit stack pointer for the privilege level 0 stack.

These MSRs can be read from and written to using the RDMSR and WRMSR instructions. The register addresses are listed in Table 3-14. These addresses are defined to remain fixed for future IA-32 processors.

**Table 3-14. MSRs Used By the SYSENTER and SYSEXIT Instructions**

MSR	Address
SYSENTER_CS_MSR	174H
SYSENTER_ESP_MSR	175H
SYSENTER_EIP_MSR	176H

When the SYSENTER instruction is executed, the processor does the following:

1. Loads the segment selector from the SYSENTER\_CS\_MSR into the CS register.
2. Loads the instruction pointer from the SYSENTER\_EIP\_MSR into the EIP register.
3. Adds 8 to the value in SYSENTER\_CS\_MSR and loads it into the SS register.
4. Loads the stack pointer from the SYSENTER\_ESP\_MSR into the ESP register.
5. Switches to privilege level 0.
6. Clears the VM flag in the EFLAGS register, if the flag is set.
7. Begins executing the selected system procedure.

## SYSENTER—Fast System Call (Continued)

The processor does not save a return IP or other state information for the calling procedure.

The SYSENTER instruction always transfers program control to a protected-mode code segment with a DPL of 0. The instruction requires that the following conditions are met by the operating system:

- The segment descriptor for the selected system code segment selects a flat, 32-bit code segment of up to 4 GBytes, with execute, read, accessed, and non-conforming permissions.
- The segment descriptor for selected system stack segment selects a flat 32-bit stack segment of up to 4 GBytes, with read, write, accessed, and expand-up permissions.

The SYSENTER 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, 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 the global descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER\_CS\_MSR MSR.
- The fast system call “stub” routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF (CPUID SEP bit is set)
    THEN IF (Family == 6) AND (Model < 3) AND (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported
        FI;
    ELSE SYSENTER/SYSEXIT_Supported
    FI;
```

## SYSENTER—Fast System Call (Continued)

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 CR0.PE = 0 THEN #GP(0); FI;

IF SYSENTER\_CS\_MSR = 0 THEN #GP(0); FI;

EFLAGS.VM  $\leftarrow$  0 (\* Insures protected mode execution \*)

EFLAGS.IF  $\leftarrow$  0 (\* Mask interrupts \*)

EFLAGS.RF  $\leftarrow$  0

CS.SEL  $\leftarrow$  SYSENTER\_CS\_MSR (\* Operating system provides CS \*)

(\* Set rest of CS to a fixed value \*)

CS.SEL.CPL  $\leftarrow$  0

CS.BASE  $\leftarrow$  0 (\* Flat segment \*)

CS.LIMIT  $\leftarrow$  FFFFH (\* 4 GByte limit \*)

CS.ARbyte.G  $\leftarrow$  1 (\* 4 KByte granularity \*)

CS.ARbyte.S  $\leftarrow$  1

CS.ARbyte.TYPE  $\leftarrow$  1011B (\* Execute + Read, Accessed \*)

CS.ARbyte.D  $\leftarrow$  1 (\* 32-bit code segment\*)

CS.ARbyte.DPL  $\leftarrow$  0

CS.ARbyte.RPL  $\leftarrow$  0

CS.ARbyte.P  $\leftarrow$  1

SS.SEL  $\leftarrow$  CS.SEL + 8

(\* Set rest of SS to a fixed value \*)

SS.BASE  $\leftarrow$  0 (\* Flat segment \*)

SS.LIMIT  $\leftarrow$  FFFFH (\* 4 GByte limit \*)

SS.ARbyte.G  $\leftarrow$  1 (\* 4 KByte granularity \*)

SS.ARbyte.S  $\leftarrow$

SS.ARbyte.TYPE  $\leftarrow$  0011B (\* Read/Write, Accessed \*)

SS.ARbyte.D  $\leftarrow$  1 (\* 32-bit stack segment\*)

SS.ARbyte.DPL  $\leftarrow$  0

SS.ARbyte.RPL  $\leftarrow$  0

SS.ARbyte.P  $\leftarrow$  1

ESP  $\leftarrow$  SYSENTER\_ESP\_MSR

EIP  $\leftarrow$  SYSENTER\_EIP\_MSR

### Flags Affected

VM, IF, RF (see Operation above)

## **SYSENTER—Fast System Call (Continued)**

### **Protected Mode Exceptions**

#GP(0)                    If SYSENTER\_CS\_MSR contains zero.

### **Real-Address Mode Exceptions**

#GP(0)                    If protected mode is not enabled.

### **Virtual-8086 Mode Exceptions**

#GP(0)                    If SYSENTER\_CS\_MSR contains zero.



## SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Description
0F 35	SYSEXIT	Fast return to privilege level 3 user code.

### Description

Executes a fast return to privilege level 3 user code. This instruction is a companion instruction to the SYSENTER instruction. The SYSEXIT 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. This instruction must be executed from code executing at privilege level 0.

Prior to executing the SYSEXIT instruction, 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:

- **SYSENTER\_CS\_MSR**—Contains the 32-bit segment selector for the privilege level 0 code segment in which the processor is currently executing. (This value is used to compute the segment selectors for the privilege level 3 code and stack segments.)
- **EDX**—Contains the 32-bit offset into the privilege level 3 code segment to the first instruction to be executed in the user code.
- **ECX**—Contains the 32-bit stack pointer for the privilege level 3 stack.

The SYSENTER\_CS\_MSR MSR can be read from and written to using the RDMSR and WRMSR instructions. The register address is listed in Table 3-14. This address is defined to remain fixed for future IA-32 processors.

When the SYSEXIT instruction is executed, the processor does the following:

1. Adds 16 to the value in SYSENTER\_CS\_MSR and loads the sum into the CS selector register.
2. Loads the instruction pointer from the EDX register into the EIP register.
3. Adds 24 to the value in SYSENTER\_CS\_MSR and loads the sum into the SS selector register.
4. Loads the stack pointer from the ECX register into the ESP register.
5. Switches to privilege level 3.
6. Begins executing the user code at the EIP address.

See “SYSENTER—Fast System Call” for information about using the SYSENTER and SYSEXIT instructions as companion call and return instructions.

## SYSEXIT—Fast Return from Fast System Call (Continued)

The SYSEXIT instruction always transfers program control to a protected-mode code segment with a DPL of 3. The instruction requires that the following conditions are met by the operating system:

- The segment descriptor for the selected user code segment selects a flat, 32-bit code segment of up to 4 GBytes, with execute, read, accessed, and non-conforming permissions.
- The segment descriptor for selected user stack segment selects a flat, 32-bit stack segment of up to 4 GBytes, with expand-up, read, write, and accessed permissions.

The SYSENTER can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF (CPUID SEP bit is set)
    THEN IF (Family == 6) AND (Model < 3) AND (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported
        FI;
    ELSE SYSENTER/SYSEXIT_Supported
FI;
```

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 SYSENTER_CS_MSR = 0 THEN #GP(0); FI;
IF CR0.PE = 0 THEN #GP(0); FI;
IF CPL ≠ 0 THEN #GP(0)
```

```
CS.SEL ← (SYSENTER_CS_MSR + 16) (* Segment selector for return CS *)
(* Set rest of CS to a fixed value *)
CS.BASE ← 0 (* Flat segment *)
CS.LIMIT ← FFFFH (* 4 GByte limit *)
CS.ARbyte.G ← 1 (* 4 KByte granularity *)
CS.ARbyte.S ← 1
CS.ARbyte.TYPE ← 1011B (* Execute, Read, Non-Conforming Code *)
CS.ARbyte.D ← 1 (* 32-bit code segment*)
CS.ARbyte.DPL ← 3
CS.ARbyte.RPL ← 3
CS.ARbyte.P ← 1
```

```
SS.SEL ← (SYSENTER_CS_MSR + 24) (* Segment selector for return SS *)
```

**SYSEXIT—Fast Return from Fast System Call (Continued)**

(\* Set rest of SS to a fixed value \*)

SS.BASE ← 0

(\* Flat segment \*)

SS.LIMIT ← FFFFH

(\* 4 GByte limit \*)

SS.ARbyte.G ← 1

(\* 4 KByte granularity \*)

SS.ARbyte.S ←

SS.ARbyte.TYPE ← 0011B

(\* Expand Up, Read/Write, Data \*)

SS.ARbyte.D ← 1

(\* 32-bit stack segment\*)

SS.ARbyte.DPL ← 3

SS.ARbyte.RPL ← 3

SS.ARbyte.P ← 1

ESP ← ECX

EIP ← EDX

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0) If SYSENTER\_CS\_MSR contains zero.

**Real-Address Mode Exceptions**

#GP(0) If protected mode is not enabled.

**Virtual-8086 Mode Exceptions**

#GP(0) If SYSENTER\_CS\_MSR contains zero.

## TEST—Logical Compare

Opcode	Instruction	Description
A8 <i>ib</i>	TEST AL, <i>imm8</i>	AND <i>imm8</i> with AL; set SF, ZF, PF according to result
A9 <i>iw</i>	TEST AX, <i>imm16</i>	AND <i>imm16</i> with AX; set SF, ZF, PF according to result
A9 <i>id</i>	TEST EAX, <i>imm32</i>	AND <i>imm32</i> with EAX; set SF, ZF, PF according to result
F6 /0 <i>ib</i>	TEST <i>r/m8</i> , <i>imm8</i>	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</i> , <i>imm16</i>	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</i> , <i>imm32</i>	AND <i>imm32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result
84 / <i>r</i>	TEST <i>r/m8</i> , <i>r8</i>	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result
85 / <i>r</i>	TEST <i>r/m16</i> , <i>r16</i>	AND <i>r16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result
85 / <i>r</i>	TEST <i>r/m32</i> , <i>r32</i>	AND <i>r32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result

### 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.

### Operation

TEMP ← SRC1 AND SRC2;

SF ← MSB(TEMP);

IF TEMP ← 0

    THEN ZF ← 1;

    ELSE ZF ← 0;

FI:

PF ← BitwiseXNOR(TEMP[0:7]);

CF ← 0;

OF ← 0;

(\*AF is Undefined\*)

### Flags Affected

The OF and CF flags are cleared 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.

**TEST—Logical Compare (Continued)**

#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.

**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.

## UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode	Instruction	Description
66 0F 2E /r	UCOMISD <i>xmm1</i> , <i>xmm2/m64</i>	Compares (unordered) the low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/m64</i> and set the EFLAGS accordingly.

### 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).

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.

### 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)

## UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS (Continued)

### SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS (Continued)

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.



## UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode	Instruction	Description
0F 2E /r	UCOMISS <i>xmm1</i> , <i>xmm2/m32</i>	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.

### Description

Performs an unordered compare of the single-precision floating-point values in the low double-words 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). In 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).

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.

### 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_ss(__m128 a, __m128 b)
int_mm_ucomilt_ss(__m128 a, __m128 b)
int_mm_ucomile_ss(__m128 a, __m128 b)
int_mm_ucomigt_ss(__m128 a, __m128 b)
int_mm_ucomige_ss(__m128 a, __m128 b)
int_mm_ucomineq_ss(__m128 a, __m128 b)
```

## UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS (Continued)

### SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

### 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.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

## UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS (Continued)

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

## UD2—Undefined Instruction

Opcode	Instruction	Description
0F 0B	UD2	Raise invalid opcode exception

### Description

Generates an invalid opcode. This instruction is provided for software testing to explicitly generate an invalid opcode. The opcode for this instruction is reserved for this purpose.

Other than raising the invalid opcode exception, this instruction is the same as the NOP instruction.

### Operation

#UD (\* Generates invalid opcode exception \*);

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD Instruction is guaranteed to raise an invalid opcode exception in all operating modes).

## UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 15 /r	UNPCKHPD <i>xmm1</i> , <i>xmm2/m128</i>	Unpacks and Interleaves double-precision floating-point values from high quadwords of <i>xmm1</i> and <i>xmm2/m128</i> .

### 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 3-18. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

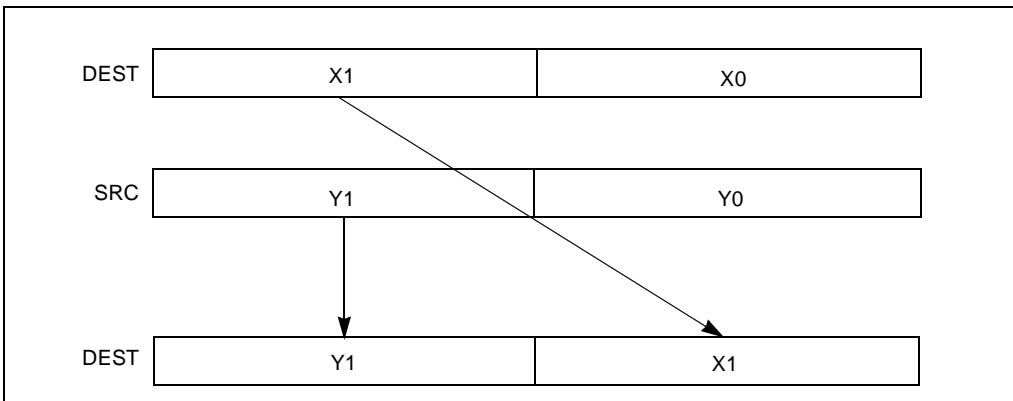


Figure 3-18. UNPCKHPD Instruction High Unpack and Interleave Operation

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.

### Operation

```
DEST[63-0] ← DEST[127-64];
DEST[127-64] ← SRC[127-64];
```

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

```
UNPCKHPD    __m128d _mm_unpackhi_pd(__m128d a, __m128d b)
```

## UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values (Continued)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## **UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

## UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 15 /r	UNPCKHPS <i>xmm1</i> , <i>xmm2/m128</i>	Unpacks and interleaves single-precision floating-point values from high quadwords of <i>xmm1</i> and <i>xmm2/mem</i> into <i>xmm1</i> .

### 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 3-19. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

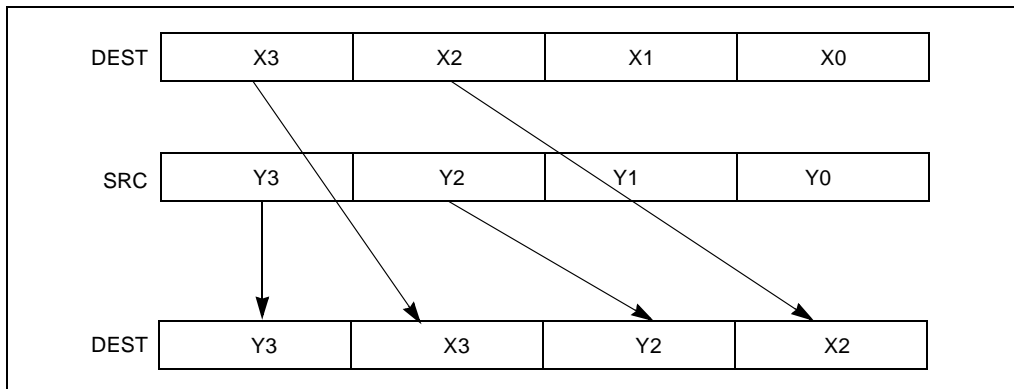


Figure 3-19. UNPCKHPS Instruction High Unpack and Interleave Operation

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.

### Operation

```
DEST[31-0] ← DEST[95-64];
DEST[63-32] ← SRC[95-64];
DEST[95-64] ← DEST[127-96];
DEST[127-96] ← SRC[127-96];
```

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

```
UNPCKHPS __m128 _mm_unpackhi_ps(__m128 a, __m128 b)
```



## UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values (Continued)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## **UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

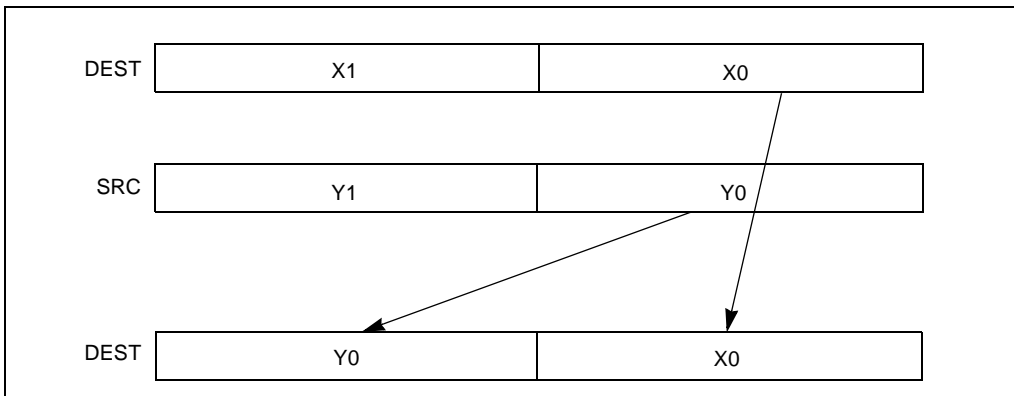
#PF(fault-code)      For a page fault.

## UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 14 /r	UNPCKLPD <i>xmm1</i> , <i>xmm2/m128</i>	Unpacks and Interleaves double-precision floating-point values from low quadwords of <i>xmm1</i> and <i>xmm2/m128</i> .

### 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 3-20. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 3-20. 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.

### Operation

```
DEST[63-0] ← DEST[63-0];
DEST[127-64] ← SRC[63-0];
```

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

```
UNPCKHPD    __m128d _mm_unpacklo_pd(__m128d a, __m128d b)
```

## UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values (Continued)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values (Continued)

### Virtual-8086 Mode Exceptions

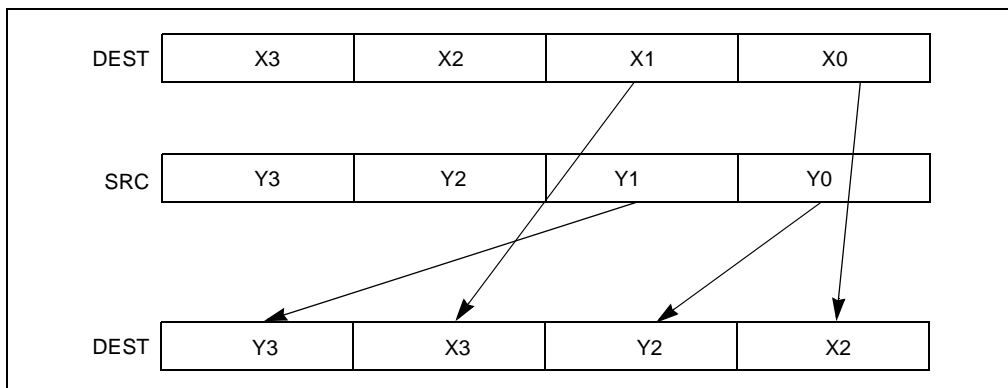
Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

## UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 14 /r	UNPCKLPS <i>xmm1</i> , <i>xmm2/m128</i>	Unpacks and Interleaves single-precision floating-point values from low quadwords of <i>xmm1</i> and <i>xmm2/mem</i> into <i>xmm1</i> .

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 3-21. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 3-21. 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.

### Operation

```
DEST[31-0] ← DEST[31-0];
DEST[63-32] ← SRC[31-0];
DEST[95-64] ← DEST[63-32];
DEST[127-96] ← SRC[63-32];
```

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

```
UNPCKLPS    __m128 __mm_unpacklo_ps(__m128 a, __m128 b)
```

## UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values (Continued)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.

## **UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values (Continued)**

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.



## VERR, VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Description
OF 00 /4	VERR <i>r/m16</i>	Set ZF=1 if segment specified with <i>r/m16</i> can be read
OF 00 /5	VERW <i>r/m16</i>	Set ZF=1 if segment specified with <i>r/m16</i> can be written

### 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.

### Operation

```
IF SRC[Offset] > (GDTR(Limit) OR (LDTR(Limit)
    THEN
        ZF ← 0
```

Read segment descriptor;

```
IF SegmentDescriptor(DescriptorType) ← 0 (* system segment *)
    OR (SegmentDescriptor(Type) ≠ conforming code segment)
    AND (CPL > DPL) OR (RPL > DPL)
    THEN
        ZF ← 0
```

## VERR, VERW—Verify a Segment for Reading or Writing (Continued)

```

ELSE
    IF ((Instruction ← VERR) AND (segment ← readable))
      OR ((Instruction ← VERW) AND (segment ← writable))
    THEN
        ZF ← 1;
    FI;
FI;

```

### Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is cleared 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.

### Real-Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real-address mode.
-----	---

### Virtual-8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode.
-----	---

**WAIT/FWAIT—Wait**

Opcode	Instruction	Description
9B	WAIT	Check pending unmasked floating-point exceptions.
9B	FWAIT	Check pending unmasked floating-point exceptions.

**Description**

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for the WAIT).

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction insures 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

**Operation**

CheckForPendingUnmaskedFloatingPointExceptions;

**FPU Flags Affected**

The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM MP and TS in CR0 is set.

**Real-Address Mode Exceptions**

#NM MP and TS in CR0 is set.

**Virtual-8086 Mode Exceptions**

#NM MP and TS in CR0 is set.

## WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Description
0F 09	WBINVD	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

### 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 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 3*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

### IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future 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.

## WBINVD—Write Back and Invalidate Cache (Continued)

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

#GP(0)                    The WBINVD instruction cannot be executed at the virtual-8086 mode.

## WRMSR—Write to Model Specific Register

Opcode	Instruction	Description
0F 30	WRMSR	Write the value in EDX:EAX to MSR specified by ECX

### Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. The input value loaded into the ECX register is the address of the MSR to be written to. 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. Undefined or reserved bits in an MSR should be set to the values previously read.

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.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated, including the global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*). (MTRRs are an implementation-specific feature of the Pentium Pro processor.)

The MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Appendix B, *Model-Specific Registers (MSRs)*, in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, 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 WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 8 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*).

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

### Operation

MSR[ECX] ← EDX:EAX;

### Flags Affected

None.

## WRMSR—Write to Model Specific Register (Continued)

### 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.

### Real-Address Mode Exceptions

- #GP                        If the value in ECX specifies a reserved or unimplemented MSR address.

### Virtual-8086 Mode Exceptions

- #GP(0)                    The WRMSR instruction is not recognized in virtual-8086 mode.

## XADD—Exchange and Add

Opcode	Instruction	Description
0F C0 /r	XADD <i>r/m8, r8</i>	Exchange <i>r8</i> and <i>r/m8</i> ; load sum into <i>r/m8</i> .
0F C1 /r	XADD <i>r/m16, r16</i>	Exchange <i>r16</i> and <i>r/m16</i> ; load sum into <i>r/m16</i> .
0F C1 /r	XADD <i>r/m32, r32</i>	Exchange <i>r32</i> and <i>r/m32</i> ; load sum into <i>r/m32</i> .

### 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.

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 nonwritable 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.



## XADD—Exchange and Add (Continued)

### 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.

## XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Description
90+rw	XCHG AX, 16	Exchange r16 with AX
90+rw	XCHG r16, X	Exchange AX with r16
90+rd	XCHG EAX, r32	Exchange r32 with EAX
90+rd	XCHG r32, EAX	Exchange EAX with r32
86/r	XCHG r/m8, r8	Exchange r8 (byte register) with byte from r/m8
86/r	XCHG r8, r/m8	Exchange byte from r/m8 with r8 (byte register)
87/r	XCHG r/m16, r16	Exchange r16 with word from r/m16
87/r	XCHG r16, r/m16	Exchange word from r/m16 with r16
87/r	XCHG r/m32, r32	Exchange r32 with doubleword from r/m32
87/r	XCHG r32, r/m32	Exchange doubleword from r/m32 with r32

### 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 7 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

### Operation

TEMP ← DEST  
 DEST ← SRC  
 SRC ← TEMP

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If either operand is in a nonwritable 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.

## XCHG—Exchange Register/Memory with Register (Continued)

#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

#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.

## XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Description
D7	XLAT <i>m8</i>	Set AL to memory byte DS:[(E)BX + unsigned AL]
D7	XLATB	Set AL to memory byte DS:[(E)BX + unsigned AL]

### 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.

### Operation

```
IF AddressSize ← 16
  THEN
    AL ← (DS:BX + ZeroExtend(AL))
  ELSE (* AddressSize ← 32 *)
    AL ← (DS:EBX + ZeroExtend(AL));
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains a null segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.

## XLAT/XLATB—Table Look-up Translation (Continued)

### 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.

## XOR—Logical Exclusive OR

Opcode	Instruction	Description
34 <i>ib</i>	XOR AL, <i>imm8</i>	AL XOR <i>imm8</i>
35 <i>iw</i>	XOR AX, <i>imm16</i>	AX XOR <i>imm16</i>
35 <i>id</i>	XOR EAX, <i>imm32</i>	EAX XOR <i>imm32</i>
80 /6 <i>ib</i>	XOR <i>r/m8</i> , <i>imm8</i>	<i>r/m8</i> XOR <i>imm8</i>
81 /6 <i>iw</i>	XOR <i>r/m16</i> , <i>imm16</i>	<i>r/m16</i> XOR <i>imm16</i>
81 /6 <i>id</i>	XOR <i>r/m32</i> , <i>imm32</i>	<i>r/m32</i> XOR <i>imm32</i>
83 /6 <i>ib</i>	XOR <i>r/m16</i> , <i>imm8</i>	<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>	<i>r/m32</i> XOR <i>imm8</i> ( <i>sign-extended</i> )
30 /r	XOR <i>r/m8</i> , <i>r8</i>	<i>r/m8</i> XOR <i>r8</i>
31 /r	XOR <i>r/m16</i> , <i>r16</i>	<i>r/m16</i> XOR <i>r16</i>
31 /r	XOR <i>r/m32</i> , <i>r32</i>	<i>r/m32</i> XOR <i>r32</i>
32 /r	XOR <i>r8</i> , <i>r/m8</i>	<i>r8</i> XOR <i>r/m8</i>
33 /r	XOR <i>r16</i> , <i>r/m16</i>	<i>r8</i> XOR <i>r/m8</i>
33 /r	XOR <i>r32</i> , <i>r/m32</i>	<i>r8</i> XOR <i>r/m8</i>

### 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.

### Operation

DEST ← DEST XOR SRC;

### Flags Affected

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

## XOR—Logical Exclusive OR (Continued)

### Protected Mode Exceptions

#GP(0)	If the destination operand points to a nonwritable 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

#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.

## XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 57 /r	XORPD <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise exclusive-OR of <i>xmm2/m128</i> and <i>xmm1</i>

### 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.

### Operation

DEST[127:0] ← DEST[127:0] BitwiseXOR SRC[127:0];

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

XORPD            \_\_m128d \_mm\_xor\_pd(\_\_m128d a, \_\_m128d b)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE2 is 0.



## XORPD—Bitwise Logical XOR of Packed Double-Precision Floating-Point Values (Continued)

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

## XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 57 /r	XORPS <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise exclusive-OR of <i>xmm2/m128</i> and <i>xmm1</i> .

### 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.

### Operation

DEST[127-0] ← DEST[127-0] BitwiseXOR SRC[127-0];

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

XORPS            \_\_m128 \_mm\_xor\_ps(\_\_m128 a, \_\_m128 b)

### 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.  If EM in CR0 is set.  If OSFXSR in CR4 is 0.  If CPUID feature flag SSE is 0.

## XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values (Continued)

### Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

**A**

# **Opcode Map**





# APPENDIX A OPCODE MAP

The opcode tables in this chapter are provided to aid in interpreting IA-32 architecture object code. The instructions are divided into three encoding groups: 1-byte opcode encodings, 2-byte opcode encodings, and escape (floating-point) encodings. The 1- and 2-byte opcode encodings are used to encode integer, system, MMX technology, SSE, and SSE2 instructions. The opcode maps for these instructions are given in Table A-2 through A-6. Section A.2.1., “One-Byte Opcode Instructions” through Section A.2.4., “Opcode Extensions For One- And Two-byte Opcodes” give instructions for interpreting 1- and 2-byte opcode maps. The escape encodings are used to encode floating-point instructions. The opcode maps for these instructions are given in Table A-7 through A-22. Section A.2.5., “Escape Opcode Instructions” gives instructions for interpreting the escape opcode maps.

The opcode tables in this section aid in interpreting IA-32 processor object code. Use the four high-order bits of the opcode as an index to a row of the opcode table; use the four low-order bits as an index to a column of the table. If the opcode is 0FH, refer to the 2-byte opcode table and use the second byte of the opcode to index the rows and columns of that table.

The escape (ESC) opcode tables for floating-point instructions identify the eight high-order bits of the opcode at the top of each page. If the accompanying ModR/M byte is in the range 00H through BFH, bits 3 through 5 identified along the top row of the third table on each page, along with the REG bits of the ModR/M, determine the opcode. ModR/M bytes outside the range 00H through BFH are mapped by the bottom two tables on each page.

Refer to Chapter 2, *Instruction Format* for detailed information on the ModR/M byte, register values, and the various addressing forms.

## A.1. 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.1.1. Codes for Addressing Method

The following abbreviations are used for addressing methods:

- A Direct address. The instruction has no ModR/M byte; the address of the operand is encoded in the instruction; and no base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).

- D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).
- E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F EFLAGS Register.
- G The reg field of the ModR/M byte selects a general register (for example, AX (000)).
- 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).
- M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).
- 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 mod field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F24, 0F26)).
- S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
- T The reg field of the ModR/M byte selects a test register (for example, MOV (0F24,0F26)).
- V The reg field of the ModR/M byte selects a 128-bit XMM register.
- W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM 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.
- X Memory addressed by the DS:SI register pair (for example, MOVS, CMPS, OUTS, or LODS).
- Y Memory addressed by the ES:DI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).



### A.1.2. Codes for Operand Type

The following abbreviations are used for 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 or 48-bit pointer, depending on operand-size attribute.
- pi Quadword MMX technology register (e.g. mm0)
- ps 128-bit packed single-precision floating-point data.
- q Quadword, regardless of operand-size attribute.
- s 6-byte pseudo-descriptor.
- ss Scalar element of a 128-bit packed single-precision floating data.
- si Doubleword integer register (e.g., eax)
- v Word or doubleword, depending on operand-size attribute.
- w Word, regardless of operand-size attribute.

### A.1.3. Register Codes

When an operand is a specific register encoded in the opcode, the register is identified by its name (for example, AX, CL, or ESI). The name of the register indicates whether the register is 32, 16, or 8 bits wide. A register identifier of the form eXX is used when the width of the register depends on the operand-size attribute. 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.

## A.2. OPCODE LOOK-UP EXAMPLES

This section provides several examples to demonstrate how the following opcode maps are used. Refer to the introduction to Chapter 3, *Instruction Set Reference*, for detailed information on the ModR/M byte, register values, and the various addressing forms.

### A.2.1. One-Byte Opcode Instructions

The opcode maps for 1-byte opcodes are shown in Table A-2 and A-3. Looking at the 1-byte opcode maps, the instruction and its operands can be determined from the hexadecimal opcode. For example:

Opcode: 030500000000H

LSB address					MSB address
03	05	00	00	00	00

Opcode 030500000000H for an ADD instruction can be interpreted from the 1-byte opcode map as follows. The first digit (0) of the opcode indicates the row, and the second digit (3) indicates the column in the opcode map tables. 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 that 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, which indicates that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3 through 5) is 000, indicating the EAX register. Thus, it can be determined that 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. These group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.2.4., “Opcode Extensions For One- And Two-byte Opcodes”).

### A.2.2. Two-Byte Opcode Instructions

Instructions that begin with 0FH can be found in the two-byte opcode maps given in Table A-4 and A-5. The second opcode byte is used to reference a particular row and column in the tables. For example, the opcode 0FA4050000000003H is located on the two-byte opcode map in row A, column 4. This opcode indicates a SHLD instruction with the operands Ev, Gv, and Ib. These operands are defined 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 indicate that a 32-bit displacement follows, located in the EAX register, and is the source.

The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H), and finally the immediate byte representing the count of the shift (03H).

By this breakdown, it has been shown that this opcode represents the instruction:

SHLD DS:00000000H, EAX, 3

The next part of the SHLD opcode is the 32-bit displacement for the destination memory operand (00000000H), which is followed by the immediate byte representing the count of the shift (03H). By this breakdown, it has been shown that the opcode 0FA4050000000003H represents the instruction:

**SHLD DS:00000000H, EAX, 3.**

Lower case is used in the following tables to highlight the mnemonics added by MMX technology, SSE, and SSE2 instructions.

### A.2.3. Opcode Map Notes

Table A-1 contains notes on particular encodings. These notes are indicated in the following Opcode Maps (Table A-2 through A-6) by superscripts.

For the One-byte Opcode Maps (Table A-2 through A-3), grey shading indicates instruction groupings.

**Table A-1. Notes on Instruction Set Encoding Tables**

Symbol	Note
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.2.4., “Opcode Extensions For One- And Two-byte Opcodes”).
1B	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions added in the Pentium III processor may 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 completely decode the instruction, see Table A-6. (These instructions include SFENCE, STMXCSR, LDMXCSR, FXRSTOR, and FXSAVE, as well as PREFETCH and its variations.)

**Table A-2. One-byte Opcode Map (Left)**

	0	1	2	3	4	5	6	7	
0	ADD						PUSH ES	POP ES	
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
1	ADC						PUSH SS	POP SS	
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
2	AND						SEG=ES	DAA	
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
3	XOR						SEG=SS	AAA	
	Eb, Gb	Ev, Gv	Gb, Eb	Gb, Ev	AL, Ib	eAX, Iv			
4	INC general register								
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
5	PUSH general register								
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
6	PUSHA/ PUSHAD	POPA/ POPAD	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS	SEG=GS	Opd Size	Addr Size	
7	Jcc, Jb - Short-displacement jump on condition								
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A	
8	Immediate Grp 1 <sup>1A</sup>				TEST		XCHG		
	Eb, Ib	Ev, Iv	Ev, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv	
9	NOP	XCHG word or double-word register with eAX						eSI	eDI
		eCX	eDX	eBX	eSP	eBP			
A	MOV								
	AL, Ob	eAX, Ov	Ob, AL	Ov, eAX	MOVS/ MOVSB Xb, Yb	MOVS/ MOVSW/ MOVSD Xv, Yv	CMPS/ CMPSB Xb, Yb	CMPS/ CMPSW/ CMPSD Xv, Yv	
B	MOV immediate byte into byte register								
	AL	CL	DL	BL	AH	CH	DH	BH	
C	Shift Grp 2 <sup>1A</sup>		RETN lw	RETN	LES Gv, Mp	LDS Gv, Mp	Grp 11 <sup>1A</sup> - MOV		
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iv	
D	Shift Grp 2 <sup>1A</sup>				AAM Ib	AAD Ib		XLAT/ XLATB	
	Eb, 1	Ev, 1	Eb, CL	Ev, CL					
E	LOOPNE/ LOOPNZ Jb	LOOPE/ LOOPZ Jb	LOOP Jb	JCXZ/ JECXZ Jb	IN		OUT		
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX	
F	LOCK		REPNE	REP/ REPE	HLT	CMC	Unary Grp 3 <sup>1A</sup>		
							Eb	Ev	

**Table A-3. One-byte Opcode Map (Right)**

8	9	A	B	C	D	E	F	
OR						PUSH CS	2-byte escape	0
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
SBB						PUSH DS	POP DS	1
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
SUB						SEG=CS	DAS	2
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
CMP						SEG=DS	AAS	3
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
DEC general register								4
eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
POP into general register								5
eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
PUSH Iv	IMUL Gv, Ev, Iv	PUSH Ib	IMUL Gv, Ev, Ib	INS/INSB Yb, DX	INS/INSW/INSD Yv, DX	OUTS/OUTSB DX, Xb	OUTS/OUTSW/OUTSD DX, Xv	6
Jcc, Jb- Short displacement jump on condition								7
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
MOV				MOV Ew, Sw	LEA Gv, M	MOV Sw, Ew	POP Ev	8
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev					
CBW/CWDE	CWD/CDQ	CALLF Ap	FWAIT/WAIT	PUSHF/PUSHFD Fv	POPF/POPFDFv	SAHF	LAHF	9
TEST		STOS/STOSB Yb, AL	STOS/STOSW/STOSD Yv, eAX	LODS/LODSB AL, Xb	LODS/LODSW/LODSD eAX, Xv	SCAS/SCASB AL, Yb	SCAS/SCASW/SCASD eAX, Xv	A
AL, Ib	eAX, Iv							
MOV immediate word or double into word or double register								B
eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
ENTER lw, lb	LEAVE	RETF lw	RETF	INT 3	INT lb	INTO	IRET	C
ESC (Escape to coprocessor instruction set)								D
CALL Jv				JMP		OUT		E
	near JV	far AP	short Jb	AL, DX	eAX, DX	DX, AL	DX, eAX	
CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 <sup>1A</sup>	INC/DEC Grp 5 <sup>1A</sup>	F

**GENERAL NOTE:**

All blanks in the opcode maps A-2 and A-3 are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

**Table A-4. Two-byte Opcode Map (Left) (First Byte is OFH)**

	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			CLTS	
1	movups Vps, Wps movss (F3) Vss, Wss	movups Wps, Vps movss (F3) Wss, Vss	movlps Wq, Vq movhlps Vq, Vq	movlps Vq, Wq	unpcklps Vps, Wq	unpckhps Vps, Wq	movhps Vq, Wq movlhps Vq, Vq	movhps Wq, Vq
2	MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		
4	CMOVcc, (Gv, Ev) - Conditional Move							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5	movmskps Ed, Vps	sqrtps Vps, Wps sqrtps (F3) Vss, Wss	rsqrtps Vps, Wps rsqrtps (F3) Vss, Wss	rcpps Vps, Wps rcpps (F3) Vss, Wss	andps Vps, Wps	andnps Vps, Wps	orps Vps, Wps	xorps Vps, Wps
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
7	pshufw Pq, Qq, lb	(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
8	Jcc, Jv - Long-displacement jump on condition							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9	SE <sub>T</sub> cc, Eb - Byte Set on condition							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A	PUSH FS	POP FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, lb	SHLD Ev, Gv, CL		
B	CMPXCHG Eb, Gb      Ev, Gv		LSS Mp	BTR Ev, Gv	LFS Mp	LGS Mp	MOVZX Gv, Eb      Gv, Ew	
C	XADD Eb, Gb	XADD Ev, Gv	cmpps Vps, Wps, lb cmpss (F3) Vss, Wss, lb		pinsrw Pq, Ed, lb	pextrw Gd, Pq, lb	shufps Vps, Wps, lb	Grp 9 <sup>1A</sup>
D		psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq		pmullw Pq, Qq		pmovmskb Gd, Pq
E	pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhw Pq, Qq	pmulhw Pq, Qq		movntq Wq, Vq
F		psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq		pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Ppi, Qpi

**GENERAL NOTE:**

All blanks in the opcode maps A-4 and A-5 are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

**Table A-5. Two-byte Opcode Map (Right) (First Byte is 0FH)**

8	9	A	B	C	D	E	F	
INVD	WBINVD		2-byte Illegal Opcodes UD2 <sup>1B</sup>					0
Prefetch <sup>1C</sup> (Grp 16 <sup>1A</sup> )								1
movaps Vps, Wps	movaps Wps, Vps	cvtpi2ps Vps, Qq cvtsi2ss (F3) Vss, Ed	movntps Wps, Vps	cvtps2pi Qq, Wps cvtsi2si (F3) Gd, Wss	cvtps2pi Qq, Wps cvtsi2si (F3) Gd, Wss	ucomiss Vss, Wss	comiss Vps, Wps	2
								3
CMOVcc(Gv, Ev) - Conditional Move								4
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
addps Vps, Wps addss (F3) Vss, Wss	mulps Vps, Wps mulss (F3) Vss, Wss			subps Vps, Wps subss (F3) Vss, Wss	minps Vps, Wps minss (F3) Vss, Wss	divps Vps, Wps divss (F3) Vss, Wss	maxps Vps, Wps maxss (F3) Vss, Wss	5
punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd Pd, Ed	movq Pq, Qq	6
MMX UD						movd Ed, Pd	movq Qq, Pq	7
Jcc, Jv - Long-displacement jump on condition								8
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
SETcc, Eb - Byte Set on condition								9
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
PUSH GS	POP GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, lb	SHRD Ev, Gv, CL	(Grp 15 <sup>1A</sup> ) <sup>1C</sup>	IMUL Gv, Ev	A
	Grp 10 <sup>1A</sup> Invalid Opcode <sup>1B</sup>	Grp 8 <sup>1A</sup> Ev, lb	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX		B
BSWAP								C
EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI	
psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq	D
psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq	E
psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq		paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq		F

### A.2.4. Opcode Extensions For One- And Two-byte Opcodes

Some of the 1-byte and 2-byte opcodes use bits 5, 4, and 3 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode. Those opcodes that have opcode extensions are indicated in Table A-6 with group numbers (Group 1, Group 2, etc.). The group numbers (ranging from 1 to A) provide an entry point into Table A-6 where the encoding of the opcode extension field can be found. For example, the ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction. Table A-6 indicates that the opcode extension that must be encoded in the ModR/M byte for this instruction is 000B.

mod	nnn	R/M
-----	-----	-----

Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)



**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number**

Opcode	Group	Mod 7,6	Encoding of Bits 5,4,3 of the ModR/M Byte							
			000	001	010	011	100	101	110	111
80-83	1	mem11	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
C0, C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem11	ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem11	TEST lb/lv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
FE	4	mem11	INC Eb	DEC Eb						
FF	5	mem11	INC Ev	DEC Ev	CALLN Ev	CALLF Ep	JMPN Ev	JMPF Ep	PUSH Ev	
OF 00	6	mem11	SLDT Ew	STR Ew	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
OF 01	7	mem11	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	INVLPG Mb
OF BA	8	mem11					BT	BTS	BTR	BTC
OF C7	9	mem		CMPXCH8 B Mq						
		11								
OF B9	10	mem								
		11								
C6	11	mem	MOV Ev, Iv							
C7		11	MOV Ev, Iv							
OF 71	12	mem								
		11			psrlw Pq, lb		psraw Pq, lb		psllw Pq, lb	
OF 72	13	mem								
		11			psrld Pq, lb		psrad Pq, lb		pslld Pq, lb	
OF 73	14	mem								
		11			psrlq Pq, lb				psllq Pq, lb	
OF AE	15	mem	fxsave	fxrstor	ldmxcsr	stmxcsr				
		11								sfence
OF 18	16	mem	prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11								

**GENERAL NOTE:**

All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

## A.2.5. Escape Opcode Instructions

The opcode maps for the escape instruction opcodes (floating-point instruction opcodes) are given in Table A-7 through A-22. These opcode maps are grouped by the first byte of the opcode from D8 through DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H through BFH, bits 5, 4, and 3 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1- and 2-byte opcodes (refer to Section A.2.4., “Opcode Extensions For One- And Two-byte Opcodes”). 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.2.5.1. OPCODES WITH MODR/M BYTES IN THE 00H THROUGH BFH RANGE

The opcode DD0504000000H can be interpreted as follows. The instruction encoded with this opcode can be located in Section A.2.5.8., “Escape Opcodes with DD as First Byte”. Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode to be for an FLD double-real instruction (refer to Table A-9). The double-real value to be loaded is at 00000004H, which is the 32-bit displacement that follows and belongs to this opcode.

### A.2.5.2. OPCODES WITH MODR/M BYTES OUTSIDE THE 00H THROUGH BFH RANGE

The opcode D8C1H illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction encoded here, can be located in Section A.2.4., “Opcode Extensions For One- And Two-byte Opcodes”. In Table A-8, the ModR/M byte C1H indicates row C, column 1, which is an FADD instruction using ST(0), ST(1) as the operands.

### A.2.5.3. ESCAPE OPCODES WITH D8 AS FIRST BYTE

Table A-7 and A-8 contain the opcodes maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (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<sup>1</sup>**

nnn Field of ModR/M Byte (refer to Figure A.2.4.)							
000	001	010	011	100	101	110	111
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

Table A-8 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

**Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH<sup>1</sup>**

	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)

**A.2.5.4. ESCAPE OPCODES WITH D9 AS FIRST BYTE**

Table A-9 and A-10 contain opcodes maps for escape instruction opcodes that begin with D9H. Table A-9 shows the opcode map if the accompanying ModR/M byte is within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the Figure A-1 nnn field) selects the instruction.

**Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH<sup>1</sup>.**

nnn Field of ModR/M Byte (refer to Figure A-1)							
000	001	010	011	100	101	110	111
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-10 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

**Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH<sup>1</sup>**

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	FXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

**A.2.5.5. ESCAPE OPCODES WITH DA AS FIRST BYTE**

Table A-11 and A-12 contain the opcodes maps for the escape instruction opcodes that begin with DAH. Table A-11 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (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<sup>1</sup>**

nnn Field of ModR/M Byte (refer to Figure A-1)							
000	001	010	011	100	101	110	111
FIADD	FIMUL	FICOM	FICOMP	FISUB	FISUBR	FIDIV	FIDIVR
dword-integer	dword-integer	dword-integer	dword-integer	dword-integer	dword-integer	dword-integer	dword-integer

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-12 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

**Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH<sup>1</sup>**

	0	1	2	3	4	5		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								

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

**A.2.5.6. ESCAPE OPCODES WITH DB AS FIRST BYTE**

Table A-13 and A-14 contain the opcodes maps for the escape instruction opcodes that begin with DBH. Table A-13 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (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<sup>1</sup>**

nnn Field of ModR/M Byte (refer to Figure A-1)							
000	001	010	011	100	101	110	111
FILD dword-integer		FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-14 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

**Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH<sup>1</sup>**

	0	1	2	3	4	5		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								

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

**A.2.5.7. ESCAPE OPCODES WITH DC AS FIRST BYTE**

Table A-15 and A-16 contain the opcodes maps for the escape instruction opcodes that begin with DCH. Table A-15 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (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<sup>1</sup>**

nnn Field of ModR/M Byte (refer to Figure A-1)							
000	001	010	011	100	101	110	111
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-16 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.



**Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH<sup>4</sup>**

	0	1	2	3	4	5		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)

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

**A.2.5.8. ESCAPE OPCODES WITH DD AS FIRST BYTE**

Table A-17 and A-18 contain the opcodes maps for the escape instruction opcodes that begin with DDH. Table A-17 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (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<sup>1</sup>**

nnn Field of ModR/M Byte (refer to Figure A-1)							
000	001	010	011	100	101	110	111
FLD double-real		FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-18 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

**Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH<sup>1</sup>**

	0	1	2	3	4	5		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								

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

**A.2.5.9. ESCAPE OPCODES WITH DE AS FIRST BYTE**

Table A-19 and A-20 contain the opcodes maps for the escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (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<sup>1</sup>**

nnn Field of ModR/M Byte (refer to Figure A-1)							
000	001	010	011	100	101	110	111
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-20 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

**Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH<sup>1</sup>**

	0	1	2	3	4	5		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		FCOMP						
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)

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

**A.2.5.10. ESCAPE OPCODES WITH DF AS FIRST BYTE**

Table A-21 and A-22 contain the opcodes maps for the escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (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<sup>1</sup>**

nnn Field of ModR/M Byte (refer to Figure A-1)							
000	001	010	011	100	101	110	111
FILD word-integer		FIST word-integer	FISTP word-integer	FBLD packed-BCD	FILD qword-integer	FBSTP packed-BCD	FISTP qword-integer

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-22 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

**Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH<sup>1</sup>**

	0	1	2	3	4	5		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								

**NOTE:**

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

# B

## **Instruction Formats and Encodings**







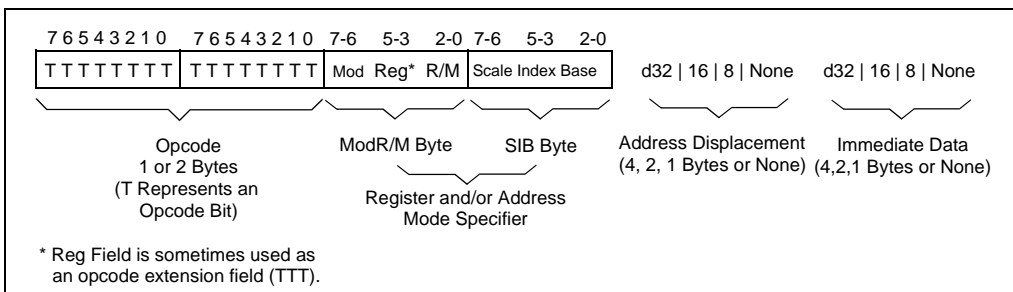
# APPENDIX B

## INSTRUCTION FORMATS AND ENCODINGS

This appendix shows the machine instruction formats and encodings of the IA-32 architecture instructions. The first section describes in detail the IA-32 architecture's machine instruction format. The following sections show the formats and encoding of the general-purpose, MMX, P6 family, SSE, SSE2, and x87 FPU instructions.

### 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 (if required) consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte, a displacement (if required), and an immediate data field (if required).



**Figure B-1. General Machine Instruction Format**

The primary opcode for an instruction is encoded in one or two bytes of the instruction. Some instructions also use an opcode extension field encoded in bits 5, 4, and 3 of the ModR/M byte. Within the primary opcode, smaller encoding fields may be defined. These fields vary according to the class of operation being performed. The fields define such information as register encoding, conditional test performed, or sign extension of immediate byte.

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, the reg field, and the R/M field. Certain encodings of the ModR/M byte indicate that a second address mode byte, the SIB byte, must be used.

If the selected addressing mode specifies a displacement, the displacement value is placed immediately following the ModR/M byte or SIB byte. If a displacement is present, the possible sizes are 8, 16, or 32 bits.

If the instruction specifies an immediate operand, the immediate value follows any displacement bytes. An immediate operand, if specified, is always the last field of the instruction.

Table B-1 lists several smaller fields or bits that appear in certain instructions, sometimes within the opcode bytes themselves. The following tables describe these fields and bits and list the allowable values. All of these fields (except the d bit) are shown in the general-purpose instruction formats given in Table B-10.

**Table B-1. Special Fields Within Instruction Encodings**

Field Name	Description	Number of Bits
reg	General-register specifier (see Table B-2 or B-3)	3
w	Specifies if data is byte or full-sized, where full-sized is either 16 or 32 bits (see Table B-4)	1
s	Specifies sign extension of an immediate data field (see Table B-5)	1
sreg2	Segment register specifier for CS, SS, DS, ES (see Table B-6)	2
sreg3	Segment register specifier for CS, SS, DS, ES, FS, GS (see Table B-6)	3
eee	Specifies a special-purpose (control or debug) register (see Table B-7)	3
ttn	For conditional instructions, specifies a condition asserted or a condition negated (see Table B-8)	4
d	Specifies direction of data operation (see Table B-9)	1

### B.1.1. Reg Field (reg)

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 (see Table B-4). Table B-2 shows the encoding of the reg field when the w bit is not present in an encoding, and 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		
Function of w Field			Function of w Field		
reg	When w = 0	When w = 1	reg	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.1.2. Encoding of Operand Size Bit (w)

The current operand-size attribute determines whether the processor is performing 16-or 32-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 (16 bits or 32 bits). Table B-4 shows the encoding of the w bit depending on the current operand-size attribute.

**Table B-4. 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.1.3. Sign Extend (s) Bit

The sign-extend (s) bit occurs primarily in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. Table B-5 shows the encoding of the s bit.

**Table B-5. 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. Segment Register Field (sreg)

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-6 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-6. 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*
		111	Reserved*

\* Do not use reserved encodings.

### B.1.5. Special-Purpose Register (eee) Field

When the control or debug registers are referenced in an instruction they are encoded in the eee field, which is located in bits 5, 4, and 3 of the ModR/M byte. Table B-7 shows the encoding of the eee field.

**Table B-7. Encoding of Special-Purpose Register (eee) Field**

eee	Control Register	Debug Register
000	CR0	DR0
001	Reserved*	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	Reserved*
101	Reserved*	Reserved*
110	Reserved*	DR6
111	Reserved*	DR7

\* Do not use reserved encodings.

### B.1.6. Condition Test Field (ttn)

For conditional instructions (such as conditional jumps and set on condition), the condition test field (ttn) is encoded for the condition being tested for. 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 ttn field is located in bits 3,2,1, and 0 of the opcode byte; for 2-byte primary opcodes, the ttn field is located in bits 3,2,1, and 0 of the second opcode byte. Table B-8 shows the encoding of the ttn field.

**Table B-8. Encoding of Conditional Test (ttn) Field**

<b>t t t n</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.1.7. Direction (d) Bit

In many two-operand instructions, a direction bit (d) indicates which operand is considered the source and which is the destination. Table B-9 shows the encoding of the d bit. When used for integer instructions, the d bit is located at bit 1 of a 1-byte primary opcode. This bit does not appear as the symbol “d” in Table B-10; instead, 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-9. Encoding of Operation Direction (d) Bit

d	Source	Destination
0	reg Field	ModR/M or SIB Byte
1	ModR/M or SIB Byte	reg Field

## B.2. GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS

Table B-10 shows the machine instruction formats and encodings of the general purpose instructions.

Table B-10. General Purpose Instruction Formats and Encodings

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

**Table B-10. General Purpose Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
memory to register	0010 001w : mod reg r/m
register to memory	0010 000w : mod reg r/m
immediate to register	
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 reg r/m
<b>BSF – Bit Scan Forward</b>	
register1, register2	0000 1111 : 1011 1100 : 11 reg2 reg1
memory, register	0000 1111 : 1011 1100 : mod reg r/m
<b>BSR – Bit Scan Reverse</b>	
register1, register2	0000 1111 : 1011 1101 : 11 reg2 reg1
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

Table B-10. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
<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>CMOVcc – Conditional Move</b>	
register2 to register1	0000 1111 : 0100 ttn : 11 reg1 reg2
memory to register	0000 1111 : 0100 ttn : mod mem r/m
<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
<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



**Table B-10. General Purpose Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<b>CMPXCHG8B – Compare and Exchange 8 Bytes</b> memory, register	0000 1111 : 1100 0111 : 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 register (alternate encoding) memory	1111 111w : 11 001 reg 0100 1 reg 1111 111w : mod 001 r/m
<b>DIV – Unsigned Divide</b> AL, AX, or EAX by register AL, AX, or EAX by memory	1111 011w : 11 110 reg 1111 011w : 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 AL, AX, or EAX by memory	1111 011w : 11 111 reg 1111 011w : mod 111 r/m
<b>IMUL – Signed Multiply</b> AL, AX, or EAX with register AL, AX, or EAX with memory register1 with register2 register with memory register1 with immediate to register2 memory with immediate to register	1111 011w : 11 101 reg 1111 011w : mod 101 reg 0000 1111 : 1010 1111 : 11 : reg1 reg2 0000 1111 : 1010 1111 : mod reg r/m 0110 10s1 : 11 reg1 reg2 : immediate data 0110 10s1 : mod reg r/m : immediate data
<b>IN – Input From Port</b> fixed port variable port	1110 010w : port number 1110 110w
<b>INC – Increment by 1</b> reg reg (alternate encoding)	1111 111w : 11 000 reg 0100 0 reg

Table B-10. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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>IRET/IRETD – Interrupt Return</b>	1100 1111
<b>Jcc – Jump if Condition is Met</b>	
8-bit displacement	0111 ttnn : 8-bit displacement
full displacement	0000 1111 : 1000 ttnn : 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 AH Register</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 reg r/m
<b>LEA – Load Effective Address</b>	1000 1101 : mod reg r/m
<b>LEAVE – High Level Procedure Exit</b>	1100 1001
<b>LES – Load Pointer to ES</b>	1100 0100 : mod reg r/m
<b>LFS – Load Pointer to FS</b>	0000 1111 : 1011 0100 : mod reg r/m
<b>LGDT – Load Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod 010 r/m
<b>LGS – Load Pointer to GS</b>	0000 1111 : 1011 0101 : mod reg r/m
<b>LIDT – Load Interrupt Descriptor Table Register</b>	

**Table B-10. General Purpose Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<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 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 : 11 000 reg
CR2 from register	0000 1111 : 0010 0010 : 11 010reg
CR3 from register	0000 1111 : 0010 0010 : 11 011 reg

Table B-10. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
CR4 from register	0000 1111 : 0010 0010 : 11 100 reg
register from CR0-CR4	0000 1111 : 0010 0000 : 11 eee reg
<b>MOV – Move to/from Debug Registers</b>	
DR0-DR3 from register	0000 1111 : 0010 0011 : 11 eee reg
DR4-DR5 from register	0000 1111 : 0010 0011 : 11 eee reg
DR6-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg
register from DR6-DR7	0000 1111 : 0010 0001 : 11 eee reg
register from DR4-DR5	0000 1111 : 0010 0001 : 11 eee reg
register from DR0-DR3	0000 1111 : 0010 0001 : 11 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>MOVSB/MOVSMB/MOVSQ/MOVSQD – Move Data from String to String</b>	1010 010w
<b>MOVSB – Move with Sign-Extend</b>	
register2 to register1	0000 1111 : 1011 111w : 11 reg1 reg2
memory to reg	0000 1111 : 1011 111w : mod reg r/m
<b>MOVZB – 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 reg
<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>NOT – One's Complement Negation</b>	
register	1111 011w : 11 010 reg

**Table B-10. General Purpose Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
memory	1111 011w : mod 010 r/m
<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>	
segment register CS, 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/POPFD – 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

Table B-10. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
<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
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>RDPMS – Read Performance Monitoring Counters</b>	0000 1111 : 0011 0011
<b>RDTS – Read Time-Stamp Counter</b>	0000 1111 : 0011 0001
<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

**Table B-10. General Purpose Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
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
<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

Table B-10. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
<b>SCAS/SCASB/SCASW/SCASD – Scan String</b>	1101 111w
<b>SETcc – Byte Set on Condition</b>	
register	0000 1111 : 1001 tttt : 11 000 reg
memory	0000 1111 : 1001 tttt : mod 000 r/m
<b>SGDT – Store Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod 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
<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 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



**Table B-10. General Purpose Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<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
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

Table B-10. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
<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
AL, 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
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. MMX INSTRUCTION FORMATS AND ENCODINGS

All MMX instructions, except the EMMS instruction, use the a format similar to the 2-byte Intel Architecture integer format. Details of subfield encodings within these formats are presented below.

#### B.3.1. 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-11 shows the encoding of this gg field.

**Table B-11. Encoding of Granularity of Data Field (gg)**

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

#### B.3.2. MMX and General-Purpose Register Fields (mmxreg and reg)

When MMX 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). Tables 2-1 and 2-2 show the 3-bit encodings used for mmxreg fields.

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. Tables 2-1 and 2-2 show the encoding of general-purpose registers when used in MMX instructions.

#### B.3.3. MMX Instruction Formats and Encodings Table

Table B-12 shows the formats and encodings of the integer instructions.

**Table B-12. MMX Instruction Formats and Encodings**

Instruction and Format	Encoding
<b>EMMS - Empty MMX state</b>	0000 1111:01110111
<b>MOVD - Move doubleword</b>	
reg to mmxreg	0000 1111:01101110: 11 mmxreg reg
reg from mmxreg	0000 1111:01111110: 11 mmxreg reg

Table B-12. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
mem to mmxreg	0000 1111:01101110: mod mmxreg r/m
mem from mmxreg	0000 1111:01111110: mod mmxreg r/m
<b>MOVQ - Move quadword</b>	
mmxreg2 to mmxreg1	0000 1111:01101111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:01111111: 11 mmxreg1 mmxreg2
mem to mmxreg	0000 1111:01101111: mod mmxreg r/m
mem from mmxreg	0000 1111:01111111: mod mmxreg r/m
<b>PACKSSDW<sup>1</sup> - Pack dword to word data (signed with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:01101011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01101011: mod mmxreg r/m
<b>PACKSSWB<sup>1</sup> - Pack word to byte data (signed with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:01100011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01100011: mod mmxreg r/m
<b>PACKUSWB<sup>1</sup> - Pack word to byte data (unsigned with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:01100111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01100111: mod mmxreg r/m
<b>PADD - Add with wrap-around</b>	
mmxreg2 to mmxreg1	0000 1111: 111111gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 111111gg: mod mmxreg r/m
<b>PADDs - Add signed with saturation</b>	
mmxreg2 to mmxreg1	0000 1111: 111011gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 111011gg: mod mmxreg r/m
<b>PADDUS - Add unsigned with saturation</b>	
mmxreg2 to mmxreg1	0000 1111: 110111gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 110111gg: mod mmxreg r/m
<b>PAND - Bitwise And</b>	
mmxreg2 to mmxreg1	0000 1111:11011011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11011011: mod mmxreg r/m
<b>PANDN - Bitwise AndNot</b>	
mmxreg2 to mmxreg1	0000 1111:11011111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11011111: mod mmxreg r/m

**Table B-12. MMX Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<b>PCMPEQ - Packed compare for equality</b> mmxreg1 with mmxreg2 mmxreg with memory	0000 1111:011101gg: 11 mmxreg1 mmxreg2 0000 1111:011101gg: mod mmxreg r/m
<b>PCMPGT - Packed compare greater (signed)</b> mmxreg1 with mmxreg2 mmxreg with memory	0000 1111:011001gg: 11 mmxreg1 mmxreg2 0000 1111:011001gg: mod mmxreg r/m
<b>PMADD - Packed multiply add</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111:11110101: 11 mmxreg1 mmxreg2 0000 1111:11110101: mod mmxreg r/m
<b>PMULH - Packed multiplication</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111:11100101: 11 mmxreg1 mmxreg2 0000 1111:11100101: mod mmxreg r/m
<b>PMULL - Packed multiplication</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111:11010101: 11 mmxreg1 mmxreg2 0000 1111:11010101: mod mmxreg r/m
<b>POR - Bitwise Or</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111:11101011: 11 mmxreg1 mmxreg2 0000 1111:11101011: mod mmxreg r/m
<b>PSLL<sup>2</sup> - Packed shift left logical</b> mmxreg1 by mmxreg2 mmxreg by memory mmxreg by immediate	0000 1111:111100gg: 11 mmxreg1 mmxreg2 0000 1111:111100gg: mod mmxreg r/m 0000 1111:011100gg: 11 110 mmxreg: imm8 data
<b>PSRA<sup>2</sup> - Packed shift right arithmetic</b> mmxreg1 by mmxreg2 mmxreg by memory mmxreg by immediate	0000 1111:111000gg: 11 mmxreg1 mmxreg2 0000 1111:111000gg: mod mmxreg r/m 0000 1111:011100gg: 11 100 mmxreg: imm8 data
<b>PSRL<sup>2</sup> - Packed shift right logical</b> mmxreg1 by mmxreg2 mmxreg by memory mmxreg by immediate	0000 1111:110100gg: 11 mmxreg1 mmxreg2 0000 1111:110100gg: mod mmxreg r/m 0000 1111:011100gg: 11 010 mmxreg: imm8 data
<b>PSUB - Subtract with wrap-around</b> mmxreg2 from mmxreg1 memory from mmxreg	0000 1111:111110gg: 11 mmxreg1 mmxreg2 0000 1111:111110gg: mod mmxreg r/m

Table B-12. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
<b>PSUBS - Subtract signed with saturation</b>	
mmxreg2 from mmxreg1	0000 1111:111010gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:111010gg: mod mmxreg r/m
<b>PSUBUS - Subtract unsigned with saturation</b>	
mmxreg2 from mmxreg1	0000 1111:110110gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:110110gg: mod mmxreg r/m
<b>PUNPCKH - Unpack high data to next larger type</b>	
mmxreg2 to mmxreg1	0000 1111:011010gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:011010gg: mod mmxreg r/m
<b>PUNPCKL - Unpack low data to next larger type</b>	
mmxreg2 to mmxreg1	0000 1111:011000gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:011000gg: mod mmxreg r/m
<b>PXOR - Bitwise Xor</b>	
mmxreg2 to mmxreg1	0000 1111:11101111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11101111: 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.4. P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS**

Table B-13 shows the formats and encodings for several instructions that were introduced into the IA-32 architecture in the P6 family processors.

Table B-13. Formats and Encodings of P6 Family Instructions

Instruction and Format	Encoding
<b>FXRSTOR—Restore x87 FPU, MMX, SSE, and SSE2 State</b>	00001111:10101110:01 m512
<b>FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State</b>	00001111:10101110:00 m512
<b>SYSENTER—Fast System Call</b>	00001111:01011111:11
<b>SYSEXIT—Fast Return from Fast System Call</b>	00001111:01011111:11

## B.5. 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-14, B-15, and B-16) show the formats and encodings for the SSE SIMD floating-point, SIMD integer, and cacheability and memory ordering instructions, respectively.

**Table B-14. Formats and Encodings of SSE SIMD Floating-Point Instructions**

Instruction and Format	Encoding
<b>ADDPS—Add Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011000: mod xmmreg r/m
<b>ADDSS—Add Scalar Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	11110011:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011000: mod xmmreg r/m
<b>ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01010101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010101: mod xmmreg r/m
<b>ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01010100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010100: mod xmmreg r/m
<b>CMPPS—Compare Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	00001111:11000010: mod xmmreg r/m: imm8
<b>CMPSS—Compare Scalar Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	11110011:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110011:00001111:11000010: mod xmmreg r/m: imm8
<b>COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg to xmmreg	00001111:00101111:11 xmmreg1 xmmreg2

Table B-14. Formats and Encodings of SSE SIMD Floating-Point Instructions

Instruction and Format	Encoding
mem to xmmreg <b>CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</b>	00001111:00101111: mod xmmreg r/m
mmreg to xmmreg	00001111:00101010:11 xmmreg1 mmreg1
mem to xmmreg <b>CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	00001111:00101010: mod xmmreg r/m
xmmreg to mmreg	00001111:00101101:11 mmreg1 xmmreg1
mem to mmreg <b>CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</b>	00001111:00101101: mod mmreg r/m
r32 to xmmreg1	11110011:00001111:00101010:11 xmmreg r32
mem to xmmreg <b>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	11110011:00001111:00101010: mod xmmreg r/m
xmmreg to r32	11110011:00001111:00101101:11 r32 xmmreg
mem to r32 <b>CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	11110011:00001111:00101101: mod r32 r/m
xmmreg to mmreg	00001111:00101100:11 mmreg1 xmmreg1
mem to mmreg <b>CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	00001111:00101100: mod mmreg r/m
xmmreg to r32	11110011:00001111:00101100:11 r32 xmmreg1
mem to r32 <b>DIVPS—Divide Packed Single-Precision Floating-Point Values</b>	11110011:00001111:00101100: mod r32 r/m
xmmreg to xmmreg	00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg <b>DIVSS—Divide Scalar Single-Precision Floating-Point Values</b>	00001111:01011110: mod xmmreg r/m
xmmreg to xmmreg	11110011:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg <b>LDMXCSR—Load MXCSR Register State</b>	11110011:00001111:01011110: mod xmmreg r/m



**Table B-14. Formats and Encodings of SSE SIMD Floating-Point Instructions**

Instruction and Format	Encoding
m32 to MXCSR	00001111:10101110:10 m32
<b>MAXPS—Return Maximum Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011111: mod xmmreg r/m
<b>MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value</b>	
xmmreg to xmmreg	11110011:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011111: mod xmmreg r/m
<b>MINPS—Return Minimum Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011101: mod xmmreg r/m
<b>MINSS—Return Minimum Scalar Double-Precision Floating-Point Value</b>	
xmmreg to xmmreg	11110011:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011101: mod xmmreg r/m
<b>MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	00001111:00101000:11 xmmreg2 xmmreg1
mem to xmmreg1	00001111:00101000: mod xmmreg r/m
xmmreg1 to xmmreg2	00001111:00101001:11 xmmreg1 xmmreg2
xmmreg1 to mem	00001111:00101001: mod xmmreg r/m
<b>MOVHPS—Move Packed Single-Precision Floating-Point Values High to Low</b>	
xmmreg to xmmreg	00001111:00010010:11 xmmreg1 xmmreg2
<b>MOVHPS—Move High Packed Single-Precision Floating-Point Values</b>	
mem to xmmreg	00001111:00010110: mod xmmreg r/m
xmmreg to mem	00001111:00010111: mod xmmreg r/m
<b>MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High</b>	
xmmreg to xmmreg	00001111:00010110:11 xmmreg1 xmmreg2
<b>MOVLPS—Move Low Packed Single-Precision Floating-Point Values</b>	
mem to xmmreg	00001111:00010010: mod xmmreg r/m

Table B-14. Formats and Encodings of SSE SIMD Floating-Point Instructions

Instruction and Format	Encoding
xmmreg to mem <b>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</b>	00001111:00010011: mod xmmreg r/m
xmmreg to r32 <b>MOVSS—Move Scalar Single-Precision Floating-Point Values</b>	00001111:01010000:11 r32 xmmreg
xmmreg2 to xmmreg1	11110011:00001111:00010000:11 xmmreg2 xmmreg1
mem to xmmreg1	11110011:00001111:00010000: mod xmmreg r/m
xmmreg1 to xmmreg2	11110011:00001111:00010000:11 xmmreg1 xmmreg2
xmmreg1 to mem <b>MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values</b>	11110011:00001111:00010000: mod xmmreg r/m
xmmreg2 to xmmreg1	00001111:00010000:11 xmmreg2 xmmreg1
mem to xmmreg1	00001111:00010000: mod xmmreg r/m
xmmreg1 to xmmreg2	00001111:00010001:11 xmmreg1 xmmreg2
xmmreg1 to mem <b>MULPS—Multiply Packed Single-Precision Floating-Point Values</b>	00001111:00010001: mod xmmreg r/m
xmmreg to xmmreg	00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg <b>MULSS—Multiply Scalar Single-Precision Floating-Point Values</b>	00001111:01011001: mod xmmreg r/m
xmmreg to xmmreg	11110011:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg <b>ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values</b>	11110011:00001111:01011001: mod xmmreg r/m
xmmreg to xmmreg	00001111:01010110:11 xmmreg1 xmmreg2
mem to xmmreg <b>RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values</b>	00001111:01010110 mod xmmreg r/m
xmmreg to xmmreg	00001111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg <b>RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value</b>	00001111:01010011: mod xmmreg r/m

**Table B-14. Formats and Encodings of SSE SIMD Floating-Point Instructions**

Instruction and Format	Encoding
xmmreg to xmmreg	11110011:00001111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01010011: mod xmmreg r/m
<b>RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01010010:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010010 mode xmmreg r/m
<b>RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value</b>	
xmmreg to xmmreg	11110011:00001111:01010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01010010 mod xmmreg r/m
<b>SHUFPS—Shuffle Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	00001111:11000110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	00001111:11000110: mod xmmreg r/m: imm8
<b>SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	00001111:01010001 mod xmmreg r/m
<b>SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value</b>	
xmmreg to xmmreg	01010011:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	01010011:00001111:01010001 mod xmmreg r/m
<b>STMXCSR—Store MXCSR Register State</b>	
MXCSR to mem	00001111:10101110:11 m32
<b>SUBPS—Subtract Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011100 mod xmmreg r/m
<b>SUBSS—Subtract Scalar Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	11110011:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011100 mod xmmreg r/m

**Table B-14. Formats and Encodings of SSE SIMD Floating-Point Instructions**

Instruction and Format	Encoding
<b>UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg to xmmreg	00001111:00101110:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00101110 mod xmmreg r/m
<b>UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:00010101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00010101 mod xmmreg r/m
<b>UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:00010100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00010100 mod xmmreg r/m
<b>XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01010111:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010111 mod xmmreg r/m

**Table B-15. Formats and Encodings of SSE SIMD Integer Instructions**

Instruction and Format	Encoding
<b>PAVGB/PAVGW—Average Packed Integers</b> mmreg to mmreg  mem to mmreg	00001111:11100000:11 mmreg1 mmreg2 00001111:11100011:11 mmreg1 mmreg2 00001111:11100000 mod mmreg r/m 00001111:11100011 mod mmreg r/m
<b>PEXTRW—Extract Word</b> mmreg to reg32, imm8	00001111:11000101:11 mmreg r32: imm8
<b>PINSRW - Insert Word</b> reg32 to mmreg, imm8 m16 to mmreg, imm8	00001111:11000100:11 r32 mmreg1: imm8 00001111:11000100 mod mmreg r/m: imm8
<b>PMAWSW—Maximum of Packed Signed Word Integers</b> mmreg to mmreg mem to mmreg	00001111:11101110:11 mmreg1 mmreg2 00001111:11101110 mod mmreg r/m
<b>PMAWSB—Maximum of Packed Unsigned Byte Integers</b> mmreg to mmreg mem to mmreg	00001111:11011110:11 mmreg1 mmreg2 00001111:11011110 mod mmreg r/m
<b>PMINSW—Minimum of Packed Signed Word Integers</b> mmreg to mmreg mem to mmreg	00001111:11101010:11 mmreg1 mmreg2 00001111:11101010 mod mmreg r/m
<b>PMINSB—Minimum of Packed Unsigned Byte Integers</b> mmreg to mmreg mem to mmreg	00001111:11011010:11 mmreg1 mmreg2 00001111:11011010 mod mmreg r/m
<b>PMOVMASKB - Move Byte Mask To Integer</b> mmreg to reg32	00001111:11010111:11 mmreg1 r32
<b>PMULHUW—Multiply Packed Unsigned Integers and Store High Result</b> mmreg to mmreg mem to mmreg	00001111:11100100:11 mmreg1 mmreg2 00001111:11100100 mod mmreg r/m
<b>PSADBW—Compute Sum of Absolute Differences</b> mmreg to mmreg mem to mmreg	00001111:11110110:11 mmreg1 mmreg2 00001111:11110110 mod mmreg r/m

**Table B-15. Formats and Encodings of SSE SIMD Integer Instructions**

Instruction and Format	Encoding
<b>PSHUFW—Shuffle Packed Words</b> mmreg to mmreg, imm8 mem to mmreg, imm8	00001111:01110000:11 mmreg1 mmreg2: imm8 00001111:01110000:11 mod mmreg r/m: imm8

**Table B-16. Format and Encoding of the SSE Cacheability and Memory Ordering Instructions**

Instruction and Format	Encoding
<b>MASKMOVQ—Store Selected Bytes of Quadword</b> mmreg to mmreg	00001111:11110111:11 mmreg1 mmreg2
<b>MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</b> xmmreg to mem	00001111:00101011 mod xmmreg r/m
<b>MOVNTQ—Store Quadword Using Non-Temporal Hint</b> mmreg to mem	00001111:11100111 mod mmreg r/m
<b>PREFETCHT0—Prefetch Temporal to All Cache Levels</b>	00001111:00011000:01 mem
<b>PREFETCHT1—Prefetch Temporal to First Level Cache</b>	00001111:00011000:10 mem
<b>PREFETCHT2—Prefetch Temporal to Second Level Cache</b>	00001111:00011000:11 mem
<b>PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels</b>	00001111:00011000:00 mem
<b>SFENCE—Store Fence</b>	00001111:10101110:11111000

## B.6. 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.

### B.6.1. 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-19 shows the encoding of this gg field.

**Table B-17. Encoding of Granularity of Data Field (gg)**

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

**Table B-18. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions**

Instruction and Format	Encoding
<b>ADDPD - Add Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011000: mod xmmreg r/m
<b>ADDSD - Add Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	11110010:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011000: mod xmmreg r/m
<b>ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01010101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010101: mod xmmreg r/m
<b>ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values</b>	

**Table B-18. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions**

Instruction and Format	Encoding
xmmreg to xmmreg	01100110:00001111:01010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010100: mod xmmreg r/m
<b>CMPPD—Compare Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	01100110:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:11000010: mod xmmreg r/m: imm8
<b>CMPSD—Compare Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	11110010:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110010:00001111:11000010: mod xmmreg r/m: imm8
<b>COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg to xmmreg	01100110:00001111:00101111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00101111: mod xmmreg r/m
<b>CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</b>	
mmreg to xmmreg	01100110:00001111:00101010:11 xmmreg1 mmreg1
mem to xmmreg	01100110:00001111:00101010: mod xmmreg r/m
<b>CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	01100110:00001111:00101101:11 mmreg1 xmmreg1
mem to mmreg	01100110:00001111:00101101: mod mmreg r/m
<b>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</b>	
r32 to xmmreg1	11110010:00001111:00101010:11 xmmreg r32
mem to xmmreg	11110010:00001111:00101010: mod xmmreg r/m
<b>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	11110010:00001111:00101101:11 r32 xmmreg
mem to r32	11110010:00001111:00101101: mod r32 r/m



**Table B-18. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions**

Instruction and Format	Encoding
<b>CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>  xmmreg to mmreg mem to mmreg	01100110:00001111:00101100:11 mmreg1 xmmreg1 01100110:00001111:00101100: mod mmreg r/m
<b>CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>  xmmreg to r32 mem to r32	11110010:00001111:00101100:11 r32 xmmreg1 11110010:00001111:00101100: mod r32 r/m
<b>CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values</b>  xmmreg to xmmreg mem to xmmreg	01100110:00001111:01011010:11 xmmreg1 xmmreg2 01100110:00001111:01011010: mod xmmreg r/m
<b>CVTSP2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values</b>  xmmreg to xmmreg mem to xmmreg	00001111:01011010:11 xmmreg1 xmmreg2 00001111:01011010: mod xmmreg r/m
<b>CVTSD2SS—Covert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value</b>  xmmreg to xmmreg mem to xmmreg	11110010:00001111:01011010:11 xmmreg1 xmmreg2 11110010:00001111:01011010: mod xmmreg r/m
<b>CVTSS2SD—Covert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value</b>  xmmreg to xmmreg mem to xmmreg	11110011:00001111:01011010:11 xmmreg1 xmmreg2 11110011:00001111:01011010: mod xmmreg r/m
<b>CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>  xmmreg to xmmreg mem to xmmreg	11110010:00001111:11100110:11 xmmreg1 xmmreg2 11110010:00001111:11100110: mod xmmreg r/m
<b>CVTTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	

**Table B-18. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions**

Instruction and Format	Encoding
xmmreg to xmmreg	01100110:00001111:11100110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100110: mod xmmreg r/m
<b>CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	11110011:00001111:11100110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:11100110: mod xmmreg r/m
<b>CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to xmmreg	01100110:00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011011: mod xmmreg r/m
<b>CVTTPS2DQ—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to xmmreg	11110011:00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011011: mod xmmreg r/m
<b>CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011011: mod xmmreg r/m
<b>DIVPD—Divide Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011110: mod xmmreg r/m
<b>DIVSD—Divide Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	11110010:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011110: mod xmmreg r/m
<b>MAXPD—Return Maximum Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011111: mod xmmreg r/m
<b>MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value</b>	
xmmreg to xmmreg	11110010:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011111: mod xmmreg r/m

**Table B-18. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions**

Instruction and Format	Encoding
<b>MINPD—Return Minimum Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011101: mod xmmreg r/m
<b>MINSD—Return Minimum Scalar Double-Precision Floating-Point Value</b>	
xmmreg to xmmreg	11110010:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011101: mod xmmreg r/m
<b>MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	01100110:00001111:00101001:11 xmmreg2 xmmreg1
mem to xmmreg1	01100110:00001111:00101001: mod xmmreg r/m
xmmreg1 to xmmreg2	01100110:00001111:00101000:11 xmmreg1 xmmreg2
xmmreg1 to mem	01100110:00001111:00101000: mod xmmreg r/m
<b>MOVHPD—Move High Packed Double-Precision Floating-Point Values</b>	
mem to xmmreg	01100110:00001111:00010111: mod xmmreg r/m
xmmreg to mem	01100110:00001111:00010110: mod xmmreg r/m
<b>MOVLPD—Move Low Packed Double-Precision Floating-Point Values</b>	
mem to xmmreg	01100110:00001111:00010011: mod xmmreg r/m
xmmreg to mem	01100110:00001111:00010010: mod xmmreg r/m
<b>MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	01100110:00001111:01010000:11 r32 xmmreg
<b>MOVSD—Move Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	11110010:00001111:00010001:11 xmmreg2 xmmreg1
mem to xmmreg1	11110010:00001111:00010001: mod xmmreg r/m
xmmreg1 to xmmreg2	11110010:00001111:00010000:11 xmmreg1 xmmreg2
xmmreg1 to mem	11110010:00001111:00010000: mod xmmreg r/m
<b>MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	01100110:00001111:00010001:11 xmmreg2 xmmreg1
mem to xmmreg1	01100110:00001111:00010001: mod xmmreg r/m
xmmreg1 to xmmreg2	01100110:00001111:00010000:11 xmmreg1 xmmreg2

Table B-18. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions

Instruction and Format	Encoding
xmmreg1 to mem	01100110:00001111:00010000: mod xmmreg r/m
<b>MULPD—Multiply Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011001: mod xmmreg rm
<b>MULSD—Multiply Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	11110010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011001: mod xmmreg r/m
<b>ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01010110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010110: mod xmmreg r/m
<b>SHUFPS—Shuffle Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	01100110:00001111:11000110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:11000110: mod xmmreg r/m: imm8
<b>SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	01100110:00001111:01010001: mod xmmreg r/m
<b>SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value</b>	
xmmreg to xmmreg	11110010:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	11110010:00001111:01010001: mod xmmreg r/m
<b>SUBPS—Subtract Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011100: mod xmmreg r/m
<b>SUBSD—Subtract Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	11110010:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011100: mod xmmreg r/m

**Table B-18. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions**

Instruction and Format	Encoding
<b>UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg to xmmreg	01100110:00001111:00101110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00101110: mod xmmreg r/m
<b>UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:00010101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00010101: mod xmmreg r/m
<b>UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:00010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00010100: mod xmmreg r/m
<b>XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	01100110:00001111:01010111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010111: mod xmmreg r/m

Table B-19. Formats and Encodings of the SSE2 SIMD Integer Instructions

Instruction and Format	Encoding
<b>MOVD - Move Doubleword</b>	
reg to xmmreg	01100110:0000 1111:01101110: 11 xmmreg reg
reg from xmmreg	01100110:0000 1111:01111110: 11 xmmreg reg
mem to xmmreg	01100110:0000 1111:01101110: mod xmmreg r/m
mem from xmmreg	01100110:0000 1111:01111110: mod xmmreg r/m
<b>MOVDQA—Move Aligned Double Quadword</b>	
xmmreg to xmmreg	01100110:00001111:01101111:11 xmmreg1 xmmreg2
	01100110:00001111:01111111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01101111: mod xmmreg r/m
mem from xmmreg	01100110:00001111:01111111: mod xmmreg r/m
<b>MOVDQU—Move Unaligned Double Quadword</b>	
xmmreg to xmmreg	11110011:00001111:01101111:11 xmmreg1 xmmreg2
	11110011:00001111:01111111:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01101111: mod xmmreg r/m
mem from xmmreg	11110011:00001111:01111111: mod xmmreg r/m
<b>MOVQ2DQ—Move Quadword from MMX to XMM Register</b>	
mmreg to xmmreg	11110011:00001111:11010110:11 mmreg1 mmreg2
<b>MOVDQ2Q—Move Quadword from XMM to MMX Register</b>	
xmmreg to mmreg	11110010:00001111:11010110:11 mmreg1 mmreg2
<b>MOVQ - Move Quadword</b>	
mmxreg2 to mmxreg1	01100110:00001111:01111110: 11 xmmreg1 xmmreg2
mmxreg2 from mmxreg1	01100110:00001111:11010110: 11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111110: mod xmmreg r/m
mem from xmmreg	01100110:00001111:11010110: mod xmmreg r/m
<b>PACKSSDW<sup>1</sup> - Pack Dword To Word Data (signed with saturation)</b>	

**Table B-19. Formats and Encodings of the SSE2 SIMD Integer Instructions**

Instruction and Format	Encoding
xmmreg2 to xmmreg1	01100110:0000 1111:01101011: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01101011: mod xmmreg r/m
<b>PACKSSWB - Pack Word To Byte Data (signed with saturation)</b>	
xmmreg2 to xmmreg1	01100110:0000 1111:01100011: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01100011: mod xmmreg r/m
<b>PACKUSWB - Pack Word To Byte Data (unsigned with saturation)</b>	
xmmreg2 to xmmreg1	01100110:0000 1111:01100111: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01100111: mod xmmreg r/m
<b>PADDQ—Add Packed Quadword Integers</b>	
mmreg to mmreg	00001111:11010100:11 mmreg1 mmreg2
mem to mmreg	00001111:11010100: mod mmreg r/m
xmmreg to xmmreg	01100110:00001111:11010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010100: mod xmmreg r/m
<b>PADD - Add With Wrap-around</b>	
xmmreg2 to xmmreg1	01100110:0000 1111: 111111gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 111111gg: mod xmmreg r/m
<b>PADDs - Add Signed With Saturation</b>	
xmmreg2 to xmmreg1	01100110:0000 1111: 111011gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 111011gg: mod xmmreg r/m
<b>PADDUS - Add Unsigned With Saturation</b>	
xmmreg2 to xmmreg1	01100110:0000 1111: 110111gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 110111gg: mod xmmreg r/m
<b>PAND - Bitwise And</b>	
xmmreg2 to xmmreg1	01100110:0000 1111:11011011: 11 xmmreg1 xmmreg2

Table B-19. Formats and Encodings of the SSE2 SIMD Integer Instructions

Instruction and Format	Encoding
memory to xmmreg <b>PANDN - Bitwise AndNot</b>	01100110:0000 1111:11011011: mod xmmreg r/m
xmmreg2 to xmmreg1	01100110:0000 1111:11011111: 11 xmmreg1 xmmreg2
memory to xmmreg <b>PAVGB—Average Packed Integers</b>	01100110:0000 1111:11011111: mod xmmreg r/m
xmmreg to xmmreg	01100110:00001111:11100000:11 xmmreg1 xmmreg2
mem to xmmreg <b>PAVGW—Average Packed Integers</b>	01100110:00001111:11100000 mod xmmreg r/m
xmmreg to xmmreg	01100110:00001111:11100011:11 xmmreg1 xmmreg2
mem to xmmreg <b>PCMPEQ - Packed Compare For Equality</b>	01100110:00001111:11100011 mod xmmreg r/m
xmmreg1 with xmmreg2	01100110:0000 1111:011101gg: 11 xmmreg1 xmmreg2
xmmreg with memory <b>PCMPGT - Packed Compare Greater (signed)</b>	01100110:0000 1111:011101gg: mod xmmreg r/m
xmmreg1 with xmmreg2	01100110:0000 1111:011001gg: 11 xmmreg1 xmmreg2
xmmreg with memory <b>PEXTRW—Extract Word</b>	01100110:0000 1111:011001gg: mod xmmreg r/m
xmmreg to reg32, imm8	01100110:00001111:11000101:11 xmmreg r32: imm8
<b>PINSRW - Insert Word</b>	
reg32 to xmmreg, imm8	01100110:00001111:11000100:11 r32 xmmreg1: imm8
m16 to xmmreg, imm8	01100110:00001111:11000100 mod xmmreg r/m: imm8
<b>PMADD - Packed Multiply Add</b>	
xmmreg2 to xmmreg1	01100110:0000 1111:11110101: 11 xmmreg1 xmmreg2
memory to xmmreg <b>PMAXSW—Maximum of Packed Signed Word Integers</b>	01100110:0000 1111:11110101: mod xmmreg r/m



**Table B-19. Formats and Encodings of the SSE2 SIMD Integer Instructions**

Instruction and Format	Encoding
xmmreg to xmmreg	01100110:00001111:11101110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110 mod xmmreg r/m
<b>PMAXUB—Maximum of Packed Unsigned Byte Integers</b>	
xmmreg to xmmreg	01100110:00001111:11011110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11011110 mod xmmreg r/m
<b>PMINSW—Minimum of Packed Signed Word Integers</b>	
xmmreg to xmmreg	01100110:00001111:11101010:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101010 mod xmmreg r/m
<b>PMINUB—Minimum of Packed Unsigned Byte Integers</b>	
xmmreg to xmmreg	01100110:00001111:11011010:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11011010 mod xmmreg r/m
<b>PMOVMSKB - Move Byte Mask To Integer</b>	
xmmreg to reg32	01100110:00001111:11010111:11 xmmreg1 r32
<b>PMULH - Packed Multiplication</b>	
xmmreg2 to xmmreg1	01100110:0000 1111:11100101: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11100101: mod xmmreg r/m
<b>PMULL - Packed Multiplication</b>	
xmmreg2 to xmmreg1	01100110:0000 1111:11010101: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11010101: mod xmmreg r/m
<b>PMULUDQ—Multiply Packed Unsigned Doubleword Integers</b>	
mmreg to mmreg	00001111:11110100:11 mmreg1 mmreg2
mem to mmreg	00001111:11110100: mod mmreg r/m
xmmreg to xmmreg	01100110:00001111:11110100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11110100: mod xmmreg r/m
<b>POR - Bitwise Or</b>	

Table B-19. Formats and Encodings of the SSE2 SIMD Integer Instructions

Instruction and Format	Encoding
xmmreg2 to xmmreg1	01100110:0000 1111:11101011: 11 xmmreg1 xmmreg2
xmemory to xmmreg	01100110:0000 1111:11101011: mod xmmreg r/m
<b>PSADBW—Compute Sum of Absolute Differences</b>	
xmmreg to xmmreg	01100110:00001111:11110110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11110110: mod xmmreg r/m
<b>PSHUFLW—Shuffle Packed Low Words</b>	
xmmreg to xmmreg, imm8	11110010:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110010:00001111:01110000:11 mod xmmreg r/m: imm8
<b>PSHUFHW—Shuffle Packed High Words</b>	
xmmreg to xmmreg, imm8	11110011:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110011:00001111:01110000:11 mod xmmreg r/m: imm8
<b>PSHUFD—Shuffle Packed Doublewords</b>	
xmmreg to xmmreg, imm8	01100110:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:01110000:11 mod xmmreg r/m: imm8
<b>PSLLDQ—Shift Double Quadword Left Logical</b>	
xmmreg, imm8	01100110:00001111:01110011:11 111 xmmreg: imm8
<b>PSLL - Packed Shift Left Logical</b>	
xmmreg1 by xmmreg2	01100110:0000 1111:111100gg: 11 xmmreg1 xmmreg2
xmmreg by memory	01100110:0000 1111:111100gg: mod xmmreg r/m
xmmreg by immediate	01100110:0000 1111:011100gg: 11 110 xmmreg: imm8 data
<b>PSRA - Packed Shift Right Arithmetic</b>	
xmmreg1 by xmmreg2	01100110:0000 1111:111100gg: 11 xmmreg1 xmmreg2
xmmreg by memory	01100110:0000 1111:111100gg: mod xmmreg r/m
xmmreg by immediate	01100110:0000 1111:011100gg: 11 100 xmmreg: imm8 data

**Table B-19. Formats and Encodings of the SSE2 SIMD Integer Instructions**

Instruction and Format	Encoding
<b>PSRLDQ—Shift Double Quadword Right Logical</b> xmmreg, imm8	01100110:00001111:01110011:11 011 xmmreg: imm8
<b>PSRL - Packed Shift Right Logical</b> xmmreg1 by xmmreg2 xmmreg by memory xmmreg by immediate	01100110:0000 1111:110100gg: 11 xmmreg1 xmmreg2 01100110:0000 1111:110100gg: mod xmmreg r/m 01100110:0000 1111:011100gg: 11 010 xmmreg: imm8 data
<b>PSUBQ—Subtract Packed Quadword Integers</b> mmreg to mmreg mem to mmreg xmmreg to xmmreg mem to xmmreg	00001111:11111011:11 mmreg1 mmreg2 00001111:11111011: mod mmreg r/m 01100110:00001111:11111011:11 xmmreg1 xmmreg2 01100110:00001111:11111011: mod xmmreg r/m
<b>PSUB - Subtract With Wrap-around</b> xmmreg2 from xmmreg1 memory from xmmreg	01100110:0000 1111:111110gg: 11 xmmreg1 xmmreg2 01100110:0000 1111:111110gg: mod xmmreg r/m
<b>PSUBS - Subtract Signed With Saturation</b> xmmreg2 from xmmreg1 memory from xmmreg	01100110:0000 1111:111010gg: 11 xmmreg1 xmmreg2 01100110:0000 1111:111010gg: mod xmmreg r/m
<b>PSUBUS - Subtract Unsigned With Saturation</b> xmmreg2 from xmmreg1 memory from xmmreg	0000 1111:110110gg: 11 xmmreg1 xmmreg2 0000 1111:110110gg: mod xmmreg r/m
<b>PUNPCKH—Unpack High Data To Next Larger Type</b> xmmreg to xmmreg mem to xmmreg	01100110:00001111:011010gg:11 xmmreg1 Xmmreg2 01100110:00001111:011010gg: mod xmmreg r/m
<b>PUNPCKHQDQ—Unpack High Data</b> xmmreg to xmmreg mem to xmmreg	01100110:00001111:01101101:11 xmmreg1 xmmreg2 01100110:00001111:01101101: mod xmmreg r/m

**Table B-19. Formats and Encodings of the SSE2 SIMD Integer Instructions**

Instruction and Format	Encoding
<b>PUNPCKL—Unpack Low Data To Next Larger Type</b> xmmreg to xmmreg	01100110:00001111:011000gg:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:011000gg: mod xmmreg r/m
<b>PUNPCKLQDQ—Unpack Low Data</b> xmmreg to xmmreg	01100110:00001111:01101100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01101100: mod xmmreg r/m
<b>PXOR - Bitwise Xor</b> xmmreg2 to xmmreg1	01100110:0000 1111:11101111: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11101111: mod xmmreg r/m

**Table B-20. Format and Encoding of the SSE2 Cacheability Instructions**

Instruction and Format	Encoding
<b>MASKMOVDQU—Store Selected Bytes of Double Quadword</b> xmmreg to xmmreg	01100110:00001111:11110111:11 xmmreg1 xmmreg2
<b>CLFLUSH—Flush Cache Line</b> mem	00001111:10101110:mod r/m
<b>MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</b> xmmreg to mem	01100110:00001111:00101011: mod xmmreg r/m
<b>MOVNTDQ—Store Double Quadword Using Non-Temporal Hint</b> xmmreg to mem	01100110:00001111:11100111: mod xmmreg r/m
<b>MOVNTI—Store Doubleword Using Non-Temporal Hint</b> reg to mem	00001111:11000011: mod reg r/m
<b>PAUSE—Spin Loop Hint</b>	11110011:10010000
<b>LFENCE—Load Fence</b>	00001111:10101110: 11 101 000
<b>MFENCE—Memory Fence</b>	00001111:10101110: 11 110 000

## B.7. FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-21 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-21. General Floating-Point Instruction Formats**

		Instruction									Optional Fields		
		First Byte			Second Byte								
1		11011	OPA		1	mod		1	OPB	r/m		s-i-b	disp
2		11011	MF		OPA	mod		OPB		r/m		s-i-b	disp
3		11011	d	P	OPA	1	1	OPB	R	ST(i)			
4		11011	0	0	1	1	1	1	OP				
5		11011	0	1	1	1	1	1	OP				
		15-11	10	9	8	7	6	5	4	3	2	1	0

MF = Memory Format  
 00 — 32-bit real  
 01 — 32-bit integer  
 10 — 64-bit real  
 11 — 16-bit integer

P = Pop  
 0 — Do not pop stack  
 1 — Pop stack after operation

d = Destination  
 0 — Destination is ST(0)  
 1 — Destination is ST(i)

R XOR d = 0 — Destination OP Source  
 R XOR d = 1 — Source OP Destination

ST(i) = Register stack element *i*  
 000 = Stack Top  
 001 = Second stack element  
 .  
 .  
 111 = Eighth stack element

The Mod and R/M fields of the ModR/M byte have the same interpretation as the corresponding fields of the integer instructions. The SIB byte and disp (displacement) are optionally present in instructions that have Mod and R/M fields. Their presence depends on the values of Mod and R/M, as for integer instructions.

Table B-22 shows the formats and encodings of the floating-point instructions.

Table B-22. Floating-Point Instruction Formats and Encodings

Instruction and Format	Encoding
<b>F2XM1 – Compute <math>2^{\text{ST}(0)} - 1</math></b>	11011 001 : 1111 0000
<b>FABS – Absolute Value</b>	11011 001 : 1110 0001
<b>FADD – Add</b>	
ST(0) ← ST(0) + 32-bit memory	11011 000 : mod 000 r/m
ST(0) ← ST(0) + 64-bit memory	11011 100 : mod 000 r/m
ST(d) ← ST(0) + ST(i)	11011 d00 : 11 000 ST(i)
<b>FADDP – Add and Pop</b>	
ST(0) ← ST(0) + ST(i)	11011 110 : 11 000 ST(i)
<b>FBLD – Load Binary Coded Decimal</b>	11011 111 : mod 100 r/m
<b>FBSTP – Store Binary Coded Decimal and Pop</b>	11011 111 : mod 110 r/m
<b>FCFS – Change Sign</b>	11011 001 : 1110 0000
<b>FCLEX – Clear Exceptions</b>	11011 011 : 1110 0010
<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)
move if not below or equal (NBE)	11011 011 : 11 010 ST(i)
move if not unordered (NU)	11011 011 : 11 011 ST(i)
<b>FCOM – Compare Real</b>	
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>FCOMI – Compare Real and Set EFLAGS</b>	11011 011 : 11 110 ST(i)
<b>FCOMIP – Compare Real, Set EFLAGS, and Pop</b>	11011 111 : 11 110 ST(i)

**Table B-22. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<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) ← ST(0) ÷ 32-bit memory	11011 000 : mod 110 r/m
ST(0) ← ST(0) ÷ 64-bit memory	11011 100 : mod 110 r/m
ST(d) ← ST(0) ÷ ST(i)	11011 d00 : 1111 R ST(i)
<b>FDIVP – Divide and Pop</b>	
ST(0) ← ST(0) ÷ ST(i)	11011 110 : 1111 1 ST(i)
<b>FDIVR – Reverse Divide</b>	
ST(0) ← 32-bit memory ÷ ST(0)	11011 000 : mod 111 r/m
ST(0) ← 64-bit memory ÷ ST(0)	11011 100 : mod 111 r/m
ST(d) ← ST(i) ÷ ST(0)	11011 d00 : 1111 R ST(i)
<b>FDIVRP – Reverse Divide and Pop</b>	
ST(0) ← ST(i) ÷ 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) ← ST(0) + 16-bit memory	11011 110 : mod 000 r/m
ST(0) ← 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</b>	
ST(0) ← ST(0) + 16-bit memory	11011 110 : mod 110 r/m
ST(0) ← ST(0) + 32-bit memory	11011 010 : mod 110 r/m
<b>FIDIVR</b>	
ST(0) ← ST(0) + 16-bit memory	11011 110 : mod 111 r/m
ST(0) ← ST(0) + 32-bit memory	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

Table B-22. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
64-bit memory	11011 111 : mod 101 r/m
<b>FIMUL</b>	
ST(0) ← ST(0) + 16-bit memory	11011 110 : mod 001 r/m
ST(0) ← ST(0) + 32-bit memory	11011 010 : mod 001 r/m
<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</b>	
ST(0) ← ST(0) + 16-bit memory	11011 110 : mod 100 r/m
ST(0) ← ST(0) + 32-bit memory	11011 010 : mod 100 r/m
<b>FISUBR</b>	
ST(0) ← ST(0) + 16-bit memory	11011 110 : mod 101 r/m
ST(0) ← ST(0) + 32-bit memory	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



**Table B-22. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<b>FMUL – Multiply</b>	
ST(0) ← ST(0) × 32-bit memory	11011 000 : mod 001 r/m
ST(0) ← ST(0) × 64-bit memory	11011 100 : mod 001 r/m
ST(d) ← ST(0) × ST(i)	11011 d00 : 1100 1 ST(i)
<b>FMULP – Multiply</b>	
ST(0) ← ST(0) × ST(i)	11011 110 : 1100 1 ST(i)
<b>FNOP – No Operation</b>	11011 001 : 1101 0000
<b>FPATAN – Partial Arc tangent</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) ← ST(0) – 32-bit memory	11011 000 : mod 100 r/m

Table B-22. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
$ST(0) \leftarrow ST(0) - 64\text{-bit memory}$	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>	
$ST(0) \leftarrow 32\text{-bit memory} - ST(0)$	11011 000 : mod 101 r/m
$ST(0) \leftarrow 64\text{-bit memory} - ST(0)$	11011 100 : mod 101 r/m
$ST(d) \leftarrow ST(i) - ST(0)$	11011 d00 : 1110 R ST(i)
<b>FSUBRP – Reverse Subtract and Pop</b>	
$ST(i) \leftarrow ST(i) - ST(0)$	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 – <math>ST(1) \times \log_2(ST(0))</math></b>	11011 001 : 1111 0001
<b>FYL2XP1 – <math>ST(1) \times \log_2(ST(0) + 1.0)</math></b>	11011 001 : 1111 1001
<b>FWAIT – Wait until FPU Ready</b>	1001 1011

intel®

C

**Intel C/C++ Compiler  
Intrinsics and  
Functional  
Equivalents**







# APPENDIX C

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

The two tables in this chapter itemize the Intel C/C++ compiler intrinsics and functional equivalents for the Intel MMX technology instructions and SSE and SSE2 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 the *Intel C/C++ Compiler User's Guide With Support for the Streaming SIMD Extensions 2* (Order Number 718195-2001). Appendix C catalogs use of these intrinsics.

The Section 3.1.3., “Intel C/C++ Compiler Intrinsics Equivalents” has more general supporting information for the following tables.

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`. Some intrinsics are “composites” because they require more than one instruction to implement them.

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:

```
127-----0
 |  2.0    | |  1.0    | |
-----
```

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:

<code>data_type</code>	Is the return data type, which can be either <code>void</code> , <code>int</code> , <code>__m64</code> , <code>__m128</code> , <code>__m128d</code> , <code>__m128i</code> . Only the <code>_mm_empty</code> intrinsic returns <code>void</code> .
<code>intrinsic_name</code>	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.
<code>parameters</code>	Represents the parameters required by each intrinsic.

## C.1. SIMPLE INTRINSICS

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
ADDPD	<code>__m128d _mm_add_pd(__m128d a, __m128d b)</code>	Adds the two DP FP (double-precision, floating-point) values of a and b.
ADDPS	<code>__m128 _mm_add_ps(__m128 a, __m128 b)</code>	Adds the four SP FP (single-precision, floating-point) values of a and b.
ADDSD	<code>__m128d _mm_add_sd(__m128d a, __m128d b)</code>	Adds the lower DP FP values of a and b; the upper three DP FP values are passed through from a.
ADDSS	<code>__m128 _mm_add_ss(__m128 a, __m128 b)</code>	Adds the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
ANDNPD	<code>__m128d _mm_andnot_pd(__m128d a, __m128d b)</code>	Computes the bitwise AND-NOT of the two DP FP values of a and b.
ANDNPS	<code>__m128 _mm_andnot_ps(__m128 a, __m128 b)</code>	Computes the bitwise AND-NOT of the four SP FP values of a and b.
ANDPD	<code>__m128d _mm_and_pd(__m128d a, __m128d b)</code>	Computes the bitwise AND of the two DP FP values of a and b.
ANDPS	<code>__m128 _mm_and_ps(__m128 a, __m128 b)</code>	Computes the bitwise AND of the four SP FP values of a and b.
CLFLUSH	<code>void _mm_clflush(void const *p)</code>	Cache line containing p is flushed and invalidated from all caches in the coherency domain.
CMPPD	<code>__m128d _mm_cmpeq_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmplt_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmple_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmpgt_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmpge_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmpneq_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmpngt_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmpnge_pd(__m128d a, __m128d b)</code>  <code>__m128d _mm_cmpord_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmpunord_pd(__m128d a, __m128d b)</code>  <code>__m128d _mm_cmpnle_pd(__m128d a, __m128d b)</code>	Compare for equality. Compare for less-than. Compare for less-than-or-equal. Compare for greater-than. Compare for greater-than-or-equal. Compare for inequality. Compare for not-less-than. Compare for not-greater-than. Compare for not-greater-than-or-equal.  Compare for ordered. Compare for unordered.  Compare for not-less-than-or-equal.

**Table C-1. Simple Intrinsics**

Mnemonic	Intrinsic	Description
CMPPS	__m128 _mm_cmpeq_ps(__m128 a, __m128 b)	Compare for equality.
	__m128 _mm_cmplt_ps(__m128 a, __m128 b)	Compare for less-than.
	__m128 _mm_cmple_ps(__m128 a, __m128 b)	Compare for less-than-or-equal.
	__m128 _mm_cmpgt_ps(__m128 a, __m128 b)	Compare for greater-than.
	__m128 _mm_cmpge_ps(__m128 a, __m128 b)	Compare for greater-than-or-equal.
	__m128 _mm_cmpneq_ps(__m128 a, __m128 b)	Compare for inequality.
	__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)	Compare for not-less-than.
	__m128 _mm_cmpngt_ps(__m128 a, __m128 b)	Compare for not-greater-than.
	__m128 _mm_cmpnge_ps(__m128 a, __m128 b)	Compare for not-greater-than-or-equal.
	__m128 _mm_cmpord_ps(__m128 a, __m128 b)	Compare for ordered.
	__m128 _mm_cmpunord_ps(__m128 a, __m128 b)	Compare for unordered.
CMPSD	__m128d _mm_cmpeq_sd(__m128d a, __m128d b)	Compare for equality.
	__m128d _mm_cmplt_sd(__m128d a, __m128d b)	Compare for less-than.
	__m128d _mm_cmple_sd(__m128d a, __m128d b)	Compare for less-than-or-equal.
	__m128d _mm_cmpgt_sd(__m128d a, __m128d b)	Compare for greater-than.
	__m128d _mm_cmpge_sd(__m128d a, __m128d b)	Compare for greater-than-or-equal.
	__m128d _mm_cmpneq_sd(__m128d a, __m128d b)	Compare for inequality.
	__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)	Compare for not-less-than.
	__m128d _mm_cmpnle_sd(__m128d a, __m128d b)	Compare for not-greater-than.
	__m128d _mm_cmpngt_sd(__m128d a, __m128d b)	Compare for not-greater-than-or-equal.
	__m128d _mm_cmpnge_sd(__m128d a, __m128d b)	Compare for ordered.
	__m128d _mm_cmpunord_sd(__m128d a, __m128d b)	Compare for unordered.
CMPSS	__m128 _mm_cmpeq_ss(__m128 a, __m128 b)	Compare for equality.
	__m128 _mm_cmplt_ss(__m128 a, __m128 b)	Compare for less-than.
	__m128 _mm_cmple_ss(__m128 a, __m128 b)	Compare for less-than-or-equal.
	__m128 _mm_cmpgt_ss(__m128 a, __m128 b)	Compare for greater-than.
	__m128 _mm_cmpge_ss(__m128 a, __m128 b)	Compare for greater-than-or-equal.
	__m128 _mm_cmpneq_ss(__m128 a, __m128 b)	Compare for inequality.
	__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)	Compare for not-less-than.



**Table C-1. Simple Intrinsics**

Mnemonic	Intrinsic	Description
	<code>__m128 __mm_cmpnle_ss(__m128 a, __m128 b)</code>	Compare for not-greater-than.
	<code>__m128 __mm_cmpngt_ss(__m128 a, __m128 b)</code>	Compare for not-greater-than-or-equal.
	<code>__m128 __mm_cmpnge_ss(__m128 a, __m128 b)</code>	Compare for ordered.
	<code>__m128 __mm_cmpord_ss(__m128 a, __m128 b)</code>	Compare for unordered.
	<code>__m128 __mm_cmpunord_ss(__m128 a, __m128 b)</code>	Compare for not-less-than-or-equal.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
COMISD	<code>int __mm_comieq_sd(__m128d a, __m128d b)</code>	Compares the lower DP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comilt_sd(__m128d a, __m128d b)</code>	Compares the lower DP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comile_sd(__m128d a, __m128d b)</code>	Compares the lower DP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comigt_sd(__m128d a, __m128d b)</code>	Compares the lower DP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comige_sd(__m128d a, __m128d b)</code>	Compares the lower DP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comineq_sd(__m128d a, __m128d b)</code>	Compares the lower SDP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.
COMISS	<code>int __mm_comieq_ss(__m128 a, __m128 b)</code>	Compares the lower SP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comilt_ss(__m128 a, __m128 b)</code>	Compares the lower SP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comile_ss(__m128 a, __m128 b)</code>	Compares the lower SP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comigt_ss(__m128 a, __m128 b)</code>	Compares the lower SP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	<code>int __mm_comige_ss(__m128 a, __m128 b)</code>	Compares the lower SP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
	<code>int __mm_comineq_ss(__m128 a, __m128 b)</code>	Compares the lower SP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.
CVTDQ2PD	<code>__m128d __mm_cvtepi32_pd(__m128i a)</code>	Convert the lower two 32-bit signed integer values in packed form in a to two DP FP values.
CVTDQ2PS	<code>__m128 __mm_cvtepi32_ps(__m128i a)</code>	Convert the four 32-bit signed integer values in packed form in a to four SP FP values.
CVTPD2DQ	<code>__m128i __mm_cvtpd_epi32(__m128d a)</code>	Convert the two DP FP values in a to two 32-bit signed integer values.
CVTPD2PI	<code>__m64 __mm_cvtpd_pi32(__m128d a)</code>	Convert the two DP FP values in a to two 32-bit signed integer values.
CVTPD2PS	<code>__m128 __mm_cvtpd_ps(__m128d a)</code>	Convert the two DP FP values in a to two SP FP values.
CVTPI2PD	<code>__m128d __mm_cvtpi32_pd(__m64 a)</code>	Convert the two 32-bit integer values in a to two DP FP values
CVTPI2PS	<code>__m128 __mm_cvt_pi2ps(__m128 a, __m64 b)</code> <code>__m128 __mm_cvtpi32_ps(__m128 a, __m64 b)</code>	Convert the two 32-bit integer values in packed form in b to two SP FP values; the upper two SP FP values are passed through from a.
CVTPS2DQ	<code>__m128i __mm_cvtps_epi32(__m128 a)</code>	Convert four SP FP values in a to four 32-bit signed integers according to the current rounding mode.
CVTPS2PD	<code>__m128d __mm_cvtps_pd(__m128 a)</code>	Convert the lower two SP FP values in a to DP FP values.
CVTPS2PI	<code>__m64 __mm_cvt_ps2pi(__m128 a)</code> <code>__m64 __mm_cvtps_pi32(__m128 a)</code>	Convert the two lower SP FP values of a to two 32-bit integers according to the current rounding mode, returning the integers in packed form.
CVTSD2SI	<code>int __mm_cvtsd_si32(__m128d a)</code>	Convert the lower DP FP value in a to a 32-bit integer value.
CVTSD2SS	<code>__m128 __mm_cvtsd_ss(__m128 a, __m128d b)</code>	Convert the lower DP FP value in b to a SP FP value; the upper three SP FP values of a are passed through.
CVTSI2SD	<code>__m128d __mm_cvtsi32_sd(__m128d a, int b)</code>	Convert the 32-bit integer value b to a DP FP value; the upper DP FP values are passed through from a.
CVTSI2SS	<code>__m128 __mm_cvt_si2ss(__m128 a, int b)</code> <code>__m128 __mm_cvtsi32_ss(__m128a, int b)</code>	Convert the 32-bit integer value b to an SP FP value; the upper three SP FP values are passed through from a.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
CVTSS2SD	<code>__m128d _mm_cvtss_sd(__m128d a, __m128 b)</code>	Convert the lower SP FP value of b to DP FP value, the upper DP FP value is passed through from a.
CVTSS2SI	<code>int _mm_cvt_ss2si(__m128 a)</code> <code>int _mm_cvtss_si32(__m128 a)</code>	Convert the lower SP FP value of a to a 32-bit integer.
CVTTPD2DQ	<code>__m128i _mm_cvttpd_epi32(__m128d a)</code>	Convert the two DP FP values of a to two 32-bit signed integer values with truncation, the upper two integer values are 0.
CVTTPD2PI	<code>__m64 _mm_cvttpd_pi32(__m128d a)</code>	Convert the two DP FP values of a to 32-bit signed integer values with truncation.
CVTTPS2DQ	<code>__m128i _mm_cvttps_epi32(__m128 a)</code>	Convert four SP FP values of a to four 32-bit integer with truncation.
CVTTPS2PI	<code>__m64 _mm_cvtt_ps2pi(__m128 a)</code> <code>__m64 _mm_cvttps_pi32(__m128 a)</code>	Convert the two lower SP FP values of a to two 32-bit integer with truncation, returning the integers in packed form.
CVTTSD2SI	<code>int _mm_cvttsd_si32(__m128d a)</code>	Convert the lower DP FP value of a to a 32-bit signed integer using truncation.

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
CVTTSS2 SI	int __mm_cvtt_ss2si(__m128 a) int __mm_cvttss_si32(__m128 a)  __m64 __mm_cvtsi32_si64(int i)	Convert the lower SP FP value of a to a 32-bit integer according to the current rounding mode.  Convert the integer object i to a 64-bit __m64 object. The integer value is zero extended to 64 bits.
	int __mm_cvtsi64_si32(__m64 m)	Convert the lower 32 bits of the __m64 object m to an integer.
DIVPD	__m128d __mm_div_pd(__m128d a, __m128d b)	Divides the two DP FP values of a and b.
DIVPS	__m128 __mm_div_ps(__m128 a, __m128 b)	Divides the four SP FP values of a and b.
DIVSD	__m128d __mm_div_sd(__m128d a, __m128d b)	Divides the lower DP FP values of a and b; the upper three DP FP values are passed through from a.
DIVSS	__m128 __mm_div_ss(__m128 a, __m128 b)	Divides the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
EMMS	void __mm_empty()	Clears the MMX technology state.
LDMXCSR	__mm_setcsr(unsigned int i)	Sets the control register to the value specified.
LFENCE	void __mm_lfence(void)	Guaranteed that every load that proceeds, in program order, the load fence instruction is globally visible before any load instruction that follows the fence in program order.
MASKMO VDQU	void __mm_maskmoveu_si128(__m128i d, __m128i n, char *p)	Conditionally store byte elements of d to address p. The high bit of each byte in the selector n determines whether the corresponding byte in d will be stored.
MASKMO VQ	void __mm_maskmove_si64(__m64 d, __m64 n, char *p)	Conditionally store byte elements of d to address p. The high bit of each byte in the selector n determines whether the corresponding byte in d will be stored.
MAXPD	__m128d __mm_max_pd(__m128d a, __m128d b)	Computes the maximums of the two DP FP values of a and b.
MAXPS	__m128 __mm_max_ps(__m128 a, __m128 b)	Computes the maximums of the four SP FP values of a and b.
MAXSD	__m128d __mm_max_sd(__m128d a, __m128d b)	Computes the maximum of the lower DP FP values of a and b; the upper DP FP values are passed through from a.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
MAXSS	<code>__m128 _mm_max_ss(__m128 a, __m128 b)</code>	Computes the maximum of the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
MFENCE	<code>void _mm_mfence(void)</code>	Guaranteed that every memory access that proceeds, in program order, the memory fence instruction is globally visible before any memory instruction that follows the fence in program order.
MINPD	<code>__m128d _mm_min_pd(__m128d a, __m128d b)</code>	Computes the minimums of the two DP FP values of a and b.
MINPS	<code>__m128 _mm_min_ps(__m128 a, __m128 b)</code>	Computes the minimums of the four SP FP values of a and b.
MINSD	<code>__m128d _mm_min_sd(__m128d a, __m128d b)</code>	Computes the minimum of the lower DP FP values of a and b; the upper DP FP values are passed through from a.
MINSS	<code>__m128 _mm_min_ss(__m128 a, __m128 b)</code>	Computes the minimum of the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
MOVAPD	<code>__m128d _mm_load_pd(double * p)</code>  <code>void _mm_store_pd(double *p, __m128d a)</code>	Loads two DP FP values. The address p must be 16-byte-aligned.  Stores two DP FP values to address p. The address p must be 16-byte-aligned.
MOVAPS	<code>__m128 _mm_load_ps(float * p)</code>  <code>void _mm_store_ps(float *p, __m128 a)</code>	Loads four SP FP values. The address p must be 16-byte-aligned.  Stores four SP FP values. The address p must be 16-byte-aligned.
MOVD	<code>__m128i _mm_cvtsi32_si128(int a)</code>  <code>int _mm_cvtsi128_si32(__m128i a)</code>  <code>__m64 _mm_cvtsi32_si64(int a)</code>  <code>int _mm_cvtsi64_si32(__m64 a)</code>	Moves 32-bit integer a to the lower 32-bit of the 128-bit destination, while zero-extending the upper bits.  Moves lower 32-bit integer of a to a 32-bit signed integer.  Moves 32-bit integer a to the lower 32-bit of the 64-bit destination, while zero-extending the upper bits.  Moves lower 32-bit integer of a to a 32-bit signed integer.
MOVDQA	<code>__m128i _mm_load_si128(__m128i * p)</code>	Loads 128-bit values from p. The address p must be 16-byte-aligned.

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
	<code>void_mm_store_si128(__m128i *p, __m128i a)</code>	Stores 128-bit value in a to address p. The address p must be 16-byte-aligned.
MOVDQU	<code>__m128i _mm_loadu_si128(__m128i * p)</code>  <code>void_mm_storeu_si128(__m128i *p, __m128i a)</code>	Loads 128-bit values from p. The address p need not be 16-byte-aligned.  Stores 128-bit value in a to address p. The address p need not be 16-byte-aligned.
MOVDQ2 Q	<code>__m64 _mm_movepi64_pi64(__m128i a)</code>	Return the lower 64-bits in a as __m64 type.
MOVHLPS	<code>__m128 _mm_movehl_ps(__m128 a, __m128 b)</code>	Moves the upper 2 SP FP values of b to the lower 2 SP FP values of the result. The upper 2 SP FP values of a are passed through to the result.
MOVHPD	<code>__m128d _mm_loadh_pd(__m128d a, double * p)</code>  <code>void_mm_storeh_pd(double * p, __m128d a)</code>	load a DP FP value from the address p to the upper 64 bits of destination; the lower 64 bits are passed through from a.  Stores the upper DP FP value of a to the address p.
MOVHPS	<code>__m128 _mm_loadh_pi(__m128 a, __m64 * p)</code>  <code>void_mm_storeh_pi(__m64 * p, __m128 a)</code>	Sets the upper two SP FP values with 64 bits of data loaded from the address p; the lower two values are passed through from a.  Stores the upper two SP FP values of a to the address p.
MOVLPD	<code>__m128d _mm_loadl_pd(__m128d a, double * p)</code>  <code>void_mm_storel_pd(double * p, __m128d a)</code>	load a DP FP value from the address p to the lower 64 bits of destination; the upper 64 bits are passed through from a.  Stores the lower DP FP value of a to the address p.
MOVLPS	<code>__m128 _mm_loadl_pi(__m128 a, __m64 *p)</code>  <code>void_mm_storel_pi(__m64 * p, __m128 a)</code>	Sets the lower two SP FP values with 64 bits of data loaded from the address p; the upper two values are passed through from a.  Stores the lower two SP FP values of a to the address p.
MOVLHPS	<code>__m128 _mm_movelh_ps(__m128 a, __m128 b)</code>	Moves the lower 2 SP FP values of b to the upper 2 SP FP values of the result. The lower 2 SP FP values of a are passed through to the result.
MOVMSK PD	<code>int _mm_movemask_pd(__m128d a)</code>	Creates a 2-bit mask from the sign bits of the two DP FP values of a.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
MOVMSK PS	int __mm_movemask_ps(__m128 a)	Creates a 4-bit mask from the most significant bits of the four SP FP values.
MOVNTD Q	void __mm_stream_si128(__m128i * p, __m128i a)	Stores the data in a to the address p without polluting the caches. If the cache line containing p is already in the cache, the cache will be updated. The address must be 16-byte-aligned.
MOVNTPD	void __mm_stream_pd(double * p, __m128d a)	Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.
MOVNTPS	void __mm_stream_ps(float * p, __m128 a)	Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.
MOVNTI	void __mm_stream_si32(int * p, int a)	Stores the data in a to the address p without polluting the caches.
MOVNTQ	void __mm_stream_pi(__m64 * p, __m64 a)	Stores the data in a to the address p without polluting the caches.
MOVQ	__m128i __mm_load_epi64(__m128i * p)  void __mm_store_epi64(__m128i * p, __m128i a)  __m128i __mm_move_epi64(__m128i a)	Loads the lower 64 bits from p into the lower 64 bits of destination and zero-extend the upper 64 bits.  Stores the lower 64 bits of a to the lower 64 bits at p.  Moves the lower 64 bits of a to the lower 64 bits of destination. The upper 64 bits are cleared.
MOVQ2D Q	__m128i __mm_movpi64_epi64(__m64 a)	Move the 64 bits of a into the lower 64-bits, while zero-extending the upper bits.
MOVSD	__m128d __mm_load_sd(double * p)  void __mm_store_sd(double * p, __m128d a)  __m128d __mm_move_sd(__m128d a, __m128d b)	Loads a DP FP value from p into the lower DP FP value and clears the upper DP FP value. The address P need not be 16-byte aligned.  Stores the lower DP FP value of a to address p. The address P need not be 16-byte aligned.  Sets the lower DP FP values of b to destination. The upper DP FP value is passed through from a.
MOVSS	__m128 __mm_load_ss(float * p)  void __mm_store_ss(float * p, __m128 a)	Loads an SP FP value into the low word and clears the upper three words.  Stores the lower SP FP value.



**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
	<code>__m128 _mm_move_ss(__m128 a, __m128 b)</code>	Sets the low word to the SP FP value of b. The upper 3 SP FP values are passed through from a.
MOVUPD	<code>__m128d _mm_loadu_pd(double * p)</code>  <code>void_mm_storeu_pd(double *p, __m128d a)</code>	Loads two DP FP values from p. The address p need not be 16-byte-aligned.  Stores two DP FP values in a to p. The address p need not be 16-byte-aligned.
MOVUPS	<code>__m128 _mm_loadu_ps(float * p)</code>  <code>void_mm_storeu_ps(float *p, __m128 a)</code>	Loads four SP FP values. The address need not be 16-byte-aligned.  Stores four SP FP values. The address need not be 16-byte-aligned.
MULPD	<code>__m128d _mm_mul_pd(__m128d a, __m128d b)</code>	Multiplies the two DP FP values of a and b.
MULPS	<code>__m128 _mm_mul_ss(__m128 a, __m128 b)</code>	Multiplies the four SP FP value of a and b.
MULSD	<code>__m128d _mm_mul_sd(__m128d a, __m128d b)</code>	Multiplies the lower DP FP value of a and b; the upper DP FP value are passed through from a.
MULSS	<code>__m128 _mm_mul_ss(__m128 a, __m128 b)</code>	Multiplies the lower SP FP value of a and b; the upper three SP FP values are passed through from a.
ORPD	<code>__m128d _mm_or_pd(__m128d a, __m128d b)</code>	Computes the bitwise OR of the two DP FP values of a and b.
ORPS	<code>__m128 _mm_or_ps(__m128 a, __m128 b)</code>	Computes the bitwise OR of the four SP FP values of a and b.
PACKSSWB	<code>__m128i _mm_packs_epi16(__m128i m1, __m128i m2)</code>	Pack the eight 16-bit values from m1 into the lower eight 8-bit values of the result with signed saturation, and pack the eight 16-bit values from m2 into the upper eight 8-bit values of the result with signed saturation.
PACKSSWB	<code>__m64 _mm_packs_pi16(__m64 m1, __m64 m2)</code>	Pack the four 16-bit values from m1 into the lower four 8-bit values of the result with signed saturation, and pack the four 16-bit values from m2 into the upper four 8-bit values of the result with signed saturation.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
PACKSSDW	<code>__m128i _mm_packs_epi32 (__m128i m1, __m128i m2)</code>	Pack the four 32-bit values from m1 into the lower four 16-bit values of the result with signed saturation, and pack the four 32-bit values from m2 into the upper four 16-bit values of the result with signed saturation.
PACKSSDQ	<code>__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)</code>	Pack the two 32-bit values from m1 into the lower two 16-bit values of the result with signed saturation, and pack the two 32-bit values from m2 into the upper two 16-bit values of the result with signed saturation.
PACKUSWB	<code>__m128i _mm_packus_epi16(__m128i m1, __m128i m2)</code>	Pack the eight 16-bit values from m1 into the lower eight 8-bit values of the result with unsigned saturation, and pack the eight 16-bit values from m2 into the upper eight 8-bit values of the result with unsigned saturation.
PACKUSWB	<code>__m64 _mm_packs_pu16(__m64 m1, __m64 m2)</code>	Pack the four 16-bit values from m1 into the lower four 8-bit values of the result with unsigned saturation, and pack the four 16-bit values from m2 into the upper four 8-bit values of the result with unsigned saturation.
PADDDB	<code>__m128i _mm_add_epi8(__m128i m1, __m128i m2)</code>	Add the 16 8-bit values in m1 to the 16 8-bit values in m2.
PADDDB	<code>__m64 _mm_add_pi8(__m64 m1, __m64 m2)</code>	Add the eight 8-bit values in m1 to the eight 8-bit values in m2.
PADDW	<code>__m128i _mm_addw_epi16(__m128i m1, __m128i m2)</code>	Add the 8 16-bit values in m1 to the 8 16-bit values in m2.
PADDW	<code>__m64 _mm_addw_pi16(__m64 m1, __m64 m2)</code>	Add the four 16-bit values in m1 to the four 16-bit values in m2.
PADDQ	<code>__m128i _mm_add_epi32(__m128i m1, __m128i m2)</code>	Add the 4 32-bit values in m1 to the 4 32-bit values in m2.
PADDQ	<code>__m64 _mm_add_pi32(__m64 m1, __m64 m2)</code>	Add the two 32-bit values in m1 to the two 32-bit values in m2.
PADDQ	<code>__m128i _mm_add_epi64(__m128i m1, __m128i m2)</code>	Add the 2 64-bit values in m1 to the 2 64-bit values in m2.
PADDQ	<code>__m64 _mm_add_si64(__m64 m1, __m64 m2)</code>	Add the 64-bit value in m1 to the 64-bit value in m2.
PADDSD	<code>__m128i _mm_adds_epi8(__m128i m1, __m128i m2)</code>	Add the 16 signed 8-bit values in m1 to the 16 signed 8-bit values in m2 and saturate.

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
PADDSB	<code>__m64 _mm_adds_pi8(__m64 m1, __m64 m2)</code>	Add the eight signed 8-bit values in m1 to the eight signed 8-bit values in m2 and saturate.
PADDSW	<code>__m128i _mm_adds_epi16(__m128i m1, __m128i m2)</code>	Add the 8 signed 16-bit values in m1 to the 8 signed 16-bit values in m2 and saturate.
PADDSW	<code>__m64 _mm_adds_pi16(__m64 m1, __m64 m2)</code>	Add the four signed 16-bit values in m1 to the four signed 16-bit values in m2 and saturate.
PADDUSB	<code>__m128i _mm_adds_epu8(__m128i m1, __m128i m2)</code>	Add the 16 unsigned 8-bit values in m1 to the 16 unsigned 8-bit values in m2 and saturate.
PADDUSB	<code>__m64 _mm_adds_pu8(__m64 m1, __m64 m2)</code>	Add the eight unsigned 8-bit values in m1 to the eight unsigned 8-bit values in m2 and saturate.
PADDUSW	<code>__m128i _mm_adds_epu16(__m128i m1, __m128i m2)</code>	Add the 8 unsigned 16-bit values in m1 to the 8 unsigned 16-bit values in m2 and saturate.
PADDUSW	<code>__m64 _mm_adds_pu16(__m64 m1, __m64 m2)</code>	Add the four unsigned 16-bit values in m1 to the four unsigned 16-bit values in m2 and saturate.
PAND	<code>__m128i _mm_and_si128(__m128i m1, __m128i m2)</code>	Perform a bitwise AND of the 128-bit value in m1 with the 128-bit value in m2.
PAND	<code>__m64 _mm_and_si64(__m64 m1, __m64 m2)</code>	Perform a bitwise AND of the 64-bit value in m1 with the 64-bit value in m2.
PANDN	<code>__m128i _mm_andnot_si128(__m128i m1, __m128i m2)</code>	Perform a logical NOT on the 128-bit value in m1 and use the result in a bitwise AND with the 128-bit value in m2.
PANDN	<code>__m64 _mm_andnot_si64(__m64 m1, __m64 m2)</code>	Perform a logical NOT on the 64-bit value in m1 and use the result in a bitwise AND with the 64-bit value in m2.
PAUSE	<code>void _mm_pause(void)</code>	The execution of the next instruction is delayed by an implementation-specific amount of time. No architectural state is modified.
PAVGB	<code>__m128i _mm_avg_epu8(__m128i a, __m128i b)</code>	Perform the packed average on the 16 8-bit values of the two operands.
PAVGB	<code>__m64 _mm_avg_pu8(__m64 a, __m64 b)</code>	Perform the packed average on the eight 8-bit values of the two operands.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
PAVGW	<code>__m128i _mm_avg_epu16(__m128i a, __m128i b)</code>	Perform the packed average on the 8 16-bit values of the two operands.
PAVGW	<code>__m64 _mm_avg_pu16(__m64 a, __m64 b)</code>	Perform the packed average on the four 16-bit values of the two operands.
PCMPEQB	<code>__m128i _mm_cmpeq_epi8(__m128i m1, __m128i m2)</code>	If the respective 8-bit values in m1 are equal to the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQB	<code>__m64 _mm_cmpeq_pi8(__m64 m1, __m64 m2)</code>	If the respective 8-bit values in m1 are equal to the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQW	<code>__m128i _mm_cmpeq_epi16 (__m128i m1, __m128i m2)</code>	If the respective 16-bit values in m1 are equal to the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQW	<code>__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)</code>	If the respective 16-bit values in m1 are equal to the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQD	<code>__m128i _mm_cmpeq_epi32(__m128i m1, __m128i m2)</code>	If the respective 32-bit values in m1 are equal to the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQD	<code>__m64 _mm_cmpeq_pi32(__m64 m1, __m64 m2)</code>	If the respective 32-bit values in m1 are equal to the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTB	<code>__m128i _mm_cmpgt_epi8 (__m128i m1, __m128i m2)</code>	If the respective 8-bit values in m1 are greater than the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTB	<code>__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)</code>	If the respective 8-bit values in m1 are greater than the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
PCMPGT W	<code>__m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)</code>	If the respective 16-bit values in m1 are greater than the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGT W	<code>__m64 _mm_cmpgt_pi16(__m64 m1, __m64 m2)</code>	If the respective 16-bit values in m1 are greater than the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTD	<code>__m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)</code>	If the respective 32-bit values in m1 are greater than the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them all to zeroes.
PCMPGTD	<code>__m64 _mm_cmpgt_pi32(__m64 m1, __m64 m2)</code>	If the respective 32-bit values in m1 are greater than the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them all to zeroes.
PEXTRW	<code>int _mm_extract_epi16(__m128i a, int n)</code>	Extracts one of the 8 words of a. The selector n must be an immediate.
PEXTRW	<code>int _mm_extract_pi16(__m64 a, int n)</code>	Extracts one of the four words of a. The selector n must be an immediate.
PINSRW	<code>__m128i _mm_insert_epi16(__m128i a, int d, int n)</code>	Inserts word d into one of 8 words of a. The selector n must be an immediate.
PINSRW	<code>__m64 _mm_insert_pi16(__m64 a, int d, int n)</code>	Inserts word d into one of four words of a. The selector n must be an immediate.
PMADDW D	<code>__m128i _mm_madd_epi16(__m128i m1, __m128i m2)</code>	Multiply 8 16-bit values in m1 by 8 16-bit values in m2 producing 8 32-bit intermediate results, which are then summed by pairs to produce 4 32-bit results.
PMADDW D	<code>__m64 _mm_madd_pi16(__m64 m1, __m64 m2)</code>	Multiply four 16-bit values in m1 by four 16-bit values in m2 producing four 32-bit intermediate results, which are then summed by pairs to produce two 32-bit results.
PMAXSW	<code>__m128i _mm_max_epi16(__m128i a, __m128i b)</code>	Computes the element-wise maximum of the 16-bit integers in a and b.
PMAXSW	<code>__m64 _mm_max_pi16(__m64 a, __m64 b)</code>	Computes the element-wise maximum of the words in a and b.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
PMAXUB	<code>__m128i _mm_max_epu8(__m128i a, __m128i b)</code>	Computes the element-wise maximum of the unsigned bytes in a and b.
PMAXUB	<code>__m64 _mm_max_pu8(__m64 a, __m64 b)</code>	Computes the element-wise maximum of the unsigned bytes in a and b.
PMINSW	<code>__m128i _mm_min_epi16(__m128i a, __m128i b)</code>	Computes the element-wise minimum of the 16-bit integers in a and b.
PMINSW	<code>__m64 _mm_min_pi16(__m64 a, __m64 b)</code>	Computes the element-wise minimum of the words in a and b.
PMINUB	<code>__m128i _mm_min_epu8(__m128i a, __m128i b)</code>	Computes the element-wise minimum of the unsigned bytes in a and b.
PMINUB	<code>__m64 _mm_min_pu8(__m64 a, __m64 b)</code>	Computes the element-wise minimum of the unsigned bytes in a and b.
PMOVMSKB	<code>int _mm_movemask_epi8(__m128i a)</code>	Creates an 16-bit mask from the most significant bits of the bytes in a.
PMOVMSKB	<code>int _mm_movemask_pi8(__m64 a)</code>	Creates an 8-bit mask from the most significant bits of the bytes in a.
PMULHUW	<code>__m128i _mm_mulhi_epu16(__m128i a, __m128i b)</code>	Multiplies the 8 unsigned words in a and b, returning the upper 16 bits of the eight 32-bit intermediate results in packed form.
PMULHUW	<code>__m64 _mm_mulhi_pu16(__m64 a, __m64 b)</code>	Multiplies the 4 unsigned words in a and b, returning the upper 16 bits of the four 32-bit intermediate results in packed form.
PMULHW	<code>__m128i _mm_mulhi_epi16(__m128i m1, __m128i m2)</code>	Multiply 8 signed 16-bit values in m1 by 8 signed 16-bit values in m2 and produce the high 16 bits of the 8 results.
PMULHW	<code>__m64 _mm_mulhi_pi16(__m64 m1, __m64 m2)</code>	Multiply four signed 16-bit values in m1 by four signed 16-bit values in m2 and produce the high 16 bits of the four results.
PMULLW	<code>__m128i _mm_mullo_epi16(__m128i m1, __m128i m2)</code>	Multiply 8 16-bit values in m1 by 8 16-bit values in m2 and produce the low 16 bits of the 8 results.
PMULLW	<code>__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)</code>	Multiply four 16-bit values in m1 by four 16-bit values in m2 and produce the low 16 bits of the four results.

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
PMULUDQ	<code>__m64 __mm_mul_su32(__m64 m1, __m64 m2)</code>	Multiply lower 32-bit unsigned value in m1 by the lower 32-bit unsigned value in m2 and store the 64 bit results.
	<code>__m128i __mm_mul_epu32(__m128i m1, __m128i m2)</code>	Multiply lower two 32-bit unsigned value in m1 by the lower two 32-bit unsigned value in m2 and store the two 64 bit results.
POR	<code>__m64 __mm_or_si64(__m64 m1, __m64 m2)</code>	Perform a bitwise OR of the 64-bit value in m1 with the 64-bit value in m2.
POR	<code>__m128i __mm_or_si128(__m128i m1, __m128i m2)</code>	Perform a bitwise OR of the 128-bit value in m1 with the 128-bit value in m2.
PREFETCH Hh	<code>void __mm_prefetch(char *a, int sel)</code>	Loads one cache line of data from address p to a location “closer” to the processor. The value sel specifies the type of prefetch operation.
PSADBW	<code>__m128i __mm_sad_epu8(__m128i a, __m128i b)</code>	Compute the absolute differences of the 16 unsigned 8-bit values of a and b; sum the upper and lower 8 differences and store the two 16-bit result into the upper and lower 64 bit.
PSADBW	<code>__m64 __mm_sad_pu8(__m64 a, __m64 b)</code>	Compute the absolute differences of the 8 unsigned 8-bit values of a and b; sum the 8 differences and store the 16-bit result, the upper 3 words are cleared.
PSHUFD	<code>__m128i __mm_shuffle_epi32(__m128i a, int n)</code>	Returns a combination of the four dwords of a. The selector n must be an immediate.
PSHUFW	<code>__m128i __mm_shufflehi_epi16(__m128i a, int n)</code>	Shuffle the upper four 16-bit words in a as specified by n. The selector n must be an immediate.
PSHUFLW	<code>__m128i __mm_shufflelo_epi16(__m128i a, int n)</code>	Shuffle the lower four 16-bit words in a as specified by n. The selector n must be an immediate.
PSHUFW	<code>__m64 __mm_shuffle_pi16(__m64 a, int n)</code>	Returns a combination of the four words of a. The selector n must be an immediate.
PSSLW	<code>__m128i __mm_sll_epi16(__m128i m, __m128i count)</code>	Shift each of 8 16-bit values in m left the amount specified by count while shifting in zeroes.
PSSLW	<code>__m128i __mm_slli_epi16(__m128i m, int count)</code>	Shift each of 8 16-bit values in m left the amount specified by count while shifting in zeroes.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
PSLLW	<code>__m64 __mm_sll_pi16(__m64 m, __m64 count)</code>	Shift four 16-bit values in <i>m</i> left the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
	<code>__m64 __mm_slli_pi16(__m64 m, int count)</code>	Shift four 16-bit values in <i>m</i> left the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSLLD	<code>__m128i __mm_slli_epi32(__m128i m, int count)</code>	Shift each of 4 32-bit values in <i>m</i> left the amount specified by <i>count</i> while shifting in zeroes.
	<code>__m128i __mm_sll_epi32(__m128i m, __m128i count)</code>	Shift each of 4 32-bit values in <i>m</i> left the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSLLD	<code>__m64 __mm_slli_pi32(__m64 m, int count)</code>	Shift two 32-bit values in <i>m</i> left the amount specified by <i>count</i> while shifting in zeroes.
	<code>__m64 __mm_sll_pi32(__m64 m, __m64 count)</code>	Shift two 32-bit values in <i>m</i> left the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSLLQ	<code>__m64 __mm_sll_si64(__m64 m, __m64 count)</code>	Shift the 64-bit value in <i>m</i> left the amount specified by <i>count</i> while shifting in zeroes.
	<code>__m64 __mm_slli_si64(__m64 m, int count)</code>	Shift the 64-bit value in <i>m</i> left the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSLLQ	<code>__m128i __mm_sll_epi64(__m128i m, __m128i count)</code>	Shift each of two 64-bit values in <i>m</i> left by the amount specified by <i>count</i> while shifting in zeroes.
	<code>__m128i __mm_slli_epi64(__m128i m, int count)</code>	Shift each of two 64-bit values in <i>m</i> left by the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSLLDQ	<code>__m128i __mm_slli_si128(__m128i m, int imm)</code>	Shift 128 bit in <i>m</i> left by <i>imm</i> bytes while shifting in zeroes.
PSRAW	<code>__m128i __mm_sra_epi16(__m128i m, __m128i count)</code>	Shift each of 8 16-bit values in <i>m</i> right the amount specified by <i>count</i> while shifting in the sign bit.



**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
	<code>__m128i _mm_srai_epi16(__m128i m, int count)</code>	Shift each of 8 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit. For the best performance, <code>count</code> should be a constant.
PSRAW	<code>__m64 _mm_sra_pi16(__m64 m, __m64 count)</code>  <code>__m64 _mm_srai_pi16(__m64 m, int count)</code>	Shift four 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit.  Shift four 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit. For the best performance, <code>count</code> should be a constant.
PSRAD	<code>__m128i _mm_sra_epi32 (__m128i m, __m128i count)</code>  <code>__m128i _mm_srai_epi32 (__m128i m, int count)</code>	Shift each of 4 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit.  Shift each of 4 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit. For the best performance, <code>count</code> should be a constant.
PSRAD	<code>__m64 _mm_sra_pi32 (__m64 m, __m64 count)</code>  <code>__m64 _mm_srai_pi32 (__m64 m, int count)</code>	Shift two 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit.  Shift two 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit. For the best performance, <code>count</code> should be a constant.
PSRLW	<code>__m128i _mm_srl_epi16 (__m128i m, __m128i count)</code>	Shift each of 8 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.
	<code>__m128i _mm_sri_epi16 (__m128i m, int count)</code>	Shift each of 8 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.
PSRLW	<code>__m64 _mm_srl_pi16 (__m64 m, __m64 count)</code>	Shift four 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.
	<code>__m64 _mm_sri_pi16(__m64 m, int count)</code>	Shift four 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSRLD	<code>__m128i _mm_srl_epi32 (__m128i m, __m128i count)</code>	Shift each of 4 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
	<code>__m128i __mm_srl_epi32 (__m128i m, int count)</code>	Shift each of 4 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSRLD	<code>__m64 __mm_srl_pi32 (__m64 m, __m64 count)</code>  <code>__m64 __mm_srl_pi32 (__m64 m, int count)</code>	Shift two 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.  Shift two 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSRLQ	<code>__m128i __mm_srl_epi64 (__m128i m, __m128i count)</code>  <code>__m128i __mm_srl_epi64 (__m128i m, int count)</code>	Shift the 2 64-bit value in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.  Shift the 2 64-bit value in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSRLQ	<code>__m64 __mm_srl_si64 (__m64 m, __m64 count)</code>  <code>__m64 __mm_srl_si64 (__m64 m, int count)</code>	Shift the 64-bit value in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.  Shift the 64-bit value in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSRLDQ	<code>__m128i __mm_srl_si128(__m128i m, int imm)</code>	Shift 128 bit in <code>m</code> right by <code>imm</code> bytes while shifting in zeroes.
PSUBB	<code>__m128i __mm_sub_epi8(__m128i m1, __m128i m2)</code>	Subtract the 16 8-bit values in <code>m2</code> from the 16 8-bit values in <code>m1</code> .
PSUBB	<code>__m64 __mm_sub_pi8(__m64 m1, __m64 m2)</code>	Subtract the eight 8-bit values in <code>m2</code> from the eight 8-bit values in <code>m1</code> .
PSUBW	<code>__m128i __mm_sub_epi16(__m128i m1, __m128i m2)</code>	Subtract the 8 16-bit values in <code>m2</code> from the 8 16-bit values in <code>m1</code> .
PSUBW	<code>__m64 __mm_sub_pi16(__m64 m1, __m64 m2)</code>	Subtract the four 16-bit values in <code>m2</code> from the four 16-bit values in <code>m1</code> .
PSUBD	<code>__m128i __mm_sub_epi32(__m128i m1, __m128i m2)</code>	Subtract the 4 32-bit values in <code>m2</code> from the 4 32-bit values in <code>m1</code> .
PSUBD	<code>__m64 __mm_sub_pi32(__m64 m1, __m64 m2)</code>	Subtract the two 32-bit values in <code>m2</code> from the two 32-bit values in <code>m1</code> .
PSUBQ	<code>__m128i __mm_sub_epi64(__m128i m1, __m128i m2)</code>	Subtract the 2 64-bit values in <code>m2</code> from the 2 64-bit values in <code>m1</code> .

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
PSUBQ	<code>__m64 _mm_sub_si64(__m64 m1, __m64 m2)</code>	Subtract the 64-bit values in m2 from the 64-bit values in m1.
PSUBSB	<code>__m128i _mm_subs_epi8(__m128i m1, __m128i m2)</code>	Subtract the 16 signed 8-bit values in m2 from the 16 signed 8-bit values in m1 and saturate.
PSUBSB	<code>__m64 _mm_subs_pi8(__m64 m1, __m64 m2)</code>	Subtract the eight signed 8-bit values in m2 from the eight signed 8-bit values in m1 and saturate.
PSUBSW	<code>__m128i _mm_subs_epi16(__m128i m1, __m128i m2)</code>	Subtract the 8 signed 16-bit values in m2 from the 8 signed 16-bit values in m1 and saturate.
PSUBSW	<code>__m64 _mm_subs_pi16(__m64 m1, __m64 m2)</code>	Subtract the four signed 16-bit values in m2 from the four signed 16-bit values in m1 and saturate.
PSUBUSB	<code>__m128i _mm_sub_epu8(__m128i m1, __m128i m2)</code>	Subtract the 16 unsigned 8-bit values in m2 from the 16 unsigned 8-bit values in m1 and saturate.
PSUBUSB	<code>__m64 _mm_sub_pu8(__m64 m1, __m64 m2)</code>	Subtract the eight unsigned 8-bit values in m2 from the eight unsigned 8-bit values in m1 and saturate.
PSUBUSW	<code>__m128i _mm_sub_epu16(__m128i m1, __m128i m2)</code>	Subtract the 8 unsigned 16-bit values in m2 from the 8 unsigned 16-bit values in m1 and saturate.
PSUBUSW	<code>__m64 _mm_sub_pu16(__m64 m1, __m64 m2)</code>	Subtract the four unsigned 16-bit values in m2 from the four unsigned 16-bit values in m1 and saturate.
PUNPCKH BW	<code>__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)</code>	Interleave the four 8-bit values from the high half of m1 with the four values from the high half of m2 and take the least significant element from m1.
PUNPCKH BW	<code>__m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)</code>	Interleave the 8 8-bit values from the high half of m1 with the 8 values from the high half of m2.
PUNPCKH WD	<code>__m64 _mm_unpackhi_pi16(__m64 m1, __m64 m2)</code>	Interleave the two 16-bit values from the high half of m1 with the two values from the high half of m2 and take the least significant element from m1.
PUNPCKH WD	<code>__m128i _mm_unpackhi_epi16(__m128i m1, __m128i m2)</code>	Interleave the 4 16-bit values from the high half of m1 with the 4 values from the high half of m2.
PUNPCKH DQ	<code>__m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)</code>	Interleave the 32-bit value from the high half of m1 with the 32-bit value from the high half of m2 and take the least significant element from m1.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
PUNPCKHDQ	<code>__m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)</code>	Interleave two 32-bit value from the high half of m1 with the two 32-bit value from the high half of m2.
PUNPCKHQDQ	<code>__m128i _mm_unpackhi_epi64(__m128i m1, __m128i m2)</code>	Interleave the 64-bit value from the high half of m1 with the 64-bit value from the high half of m2.
PUNPCKLBW	<code>__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)</code>	Interleave the four 8-bit values from the low half of m1 with the four values from the low half of m2 and take the least significant element from m1.
PUNPCKLBW	<code>__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)</code>	Interleave the 8 8-bit values from the low half of m1 with the 8 values from the low half of m2.
PUNPCKLWD	<code>__m64 _mm_unpacklo_pi16(__m64 m1, __m64 m2)</code>	Interleave the two 16-bit values from the low half of m1 with the two values from the low half of m2 and take the least significant element from m1.
PUNPCKLWD	<code>__m128i _mm_unpacklo_epi16(__m128i m1, __m128i m2)</code>	Interleave the 4 16-bit values from the low half of m1 with the 4 values from the low half of m2.
PUNPCKLDQ	<code>__m64 _mm_unpacklo_pi32(__m64 m1, __m64 m2)</code>	Interleave the 32-bit value from the low half of m1 with the 32-bit value from the low half of m2 and take the least significant element from m1.
PUNPCKLDQ	<code>__m128i _mm_unpacklo_epi32(__m128i m1, __m128i m2)</code>	Interleave two 32-bit value from the low half of m1 with the two 32-bit value from the low half of m2.
PUNPCKHQDQ	<code>__m128i _mm_unpacklo_epi64(__m128i m1, __m128i m2)</code>	Interleave the 64-bit value from the low half of m1 with the 64-bit value from the low half of m2.
PXOR	<code>__m64 _mm_xor_si64(__m64 m1, __m64 m2)</code>	Perform a bitwise XOR of the 64-bit value in m1 with the 64-bit value in m2.
PXOR	<code>__m128i _mm_xor_si128(__m128i m1, __m128i m2)</code>	Perform a bitwise XOR of the 128-bit value in m1 with the 128-bit value in m2.
RCPPS	<code>__m128 _mm_rcp_ps(__m128 a)</code>	Computes the approximations of the reciprocals of the four SP FP values of a.
RCPSS	<code>__m128 _mm_rcp_ss(__m128 a)</code>	Computes the approximation of the reciprocal of the lower SP FP value of a; the upper three SP FP values are passed through.

**Table C-1. Simple Intrinsics**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
RSQRTPS	<code>__m128 _mm_rsqrtps(__m128 a)</code>	Computes the approximations of the reciprocals of the square roots of the four SP FP values of a.
RSQRTSS	<code>__m128 _mm_rsqrtss(__m128 a)</code>	Computes the approximation of the reciprocal of the square root of the lower SP FP value of a; the upper three SP FP values are passed through.
SFENCE	<code>void _mm_sfence(void)</code>	Guarantees that every preceding store is globally visible before any subsequent store.
SHUFPS	<code>__m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)</code>	Selects two specific DP FP values from a and b, based on the mask imm8. The mask must be an immediate.
SHUFPS	<code>__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)</code>	Selects four specific SP FP values from a and b, based on the mask imm8. The mask must be an immediate.
SQRTPD	<code>__m128d _mm_sqrtpd(__m128d a)</code>	Computes the square roots of the two DP FP values of a.
SQRTPS	<code>__m128 _mm_sqrtps(__m128 a)</code>	Computes the square roots of the four SP FP values of a.
SQRTSD	<code>__m128d _mm_sqrt_sd(__m128d a)</code>	Computes the square root of the lower DP FP value of a; the upper DP FP values are passed through.
SQRTSS	<code>__m128 _mm_sqrt_ss(__m128 a)</code>	Computes the square root of the lower SP FP value of a; the upper three SP FP values are passed through.
STMXCSR	<code>_mm_getcsr(void)</code>	Returns the contents of the control register.
SUBPD	<code>__m128d _mm_sub_pd(__m128d a, __m128d b)</code>	Subtracts the two DP FP values of a and b.
SUBPS	<code>__m128 _mm_sub_ps(__m128 a, __m128 b)</code>	Subtracts the four SP FP values of a and b.
SUBSD	<code>__m128d _mm_sub_sd(__m128d a, __m128d b)</code>	Subtracts the lower DP FP values of a and b. The upper DP FP values are passed through from a.
SUBSS	<code>__m128 _mm_sub_ss(__m128 a, __m128 b)</code>	Subtracts the lower SP FP values of a and b. The upper three SP FP values are passed through from a.

**Table C-1. Simple Intrinsics**

Mnemonic	Intrinsic	Description
UCOMISD	int _mm_ucomieq_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomilt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomile_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomigt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomige_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomineq_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.
UCOMISS	int _mm_ucomieq_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomilt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomile_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomigt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.

**Table C-1. Simple Intrinsics**

Mnemonic	Intrinsic	Description
	<code>int __mm_ucomige_ss(__m128 a, __m128 b)</code>  <code>int __mm_ucomineq_ss(__m128 a, __m128 b)</code>	<p>Compares the lower SP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.</p> <p>Compares the lower SP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.</p>
UNPCKHP D	<code>__m128d __mm_unpackhi_pd(__m128d a, __m128d b)</code>	Selects and interleaves the upper DP FP values from a and b.
UNPCKHP S	<code>__m128 __mm_unpackhi_ps(__m128 a, __m128 b)</code>	Selects and interleaves the upper two SP FP values from a and b.
UNPCKLP D	<code>__m128d __mm_unpacklo_pd(__m128d a, __m128d b)</code>	Selects and interleaves the lower DP FP values from a and b.
UNPCKLP S	<code>__m128 __mm_unpacklo_ps(__m128 a, __m128 b)</code>	Selects and interleaves the lower two SP FP values from a and b.
XORPD	<code>__m128d __mm_xor_pd(__m128d a, __m128d b)</code>	Computes bitwise EXOR (exclusive-or) of the two DP FP values of a and b.
XORPS	<code>__m128 __mm_xor_ps(__m128 a, __m128 b)</code>	Computes bitwise EXOR (exclusive-or) of the four SP FP values of a and b.

## C.2. COMPOSITE INTRINSICS

Table C-2. Composite Intrinsics

Mnemonic	Intrinsic	Description
(composite)	<code>__m128i _mm_set_epi64(__m64 q1, __m64 q0)</code>	Sets the two 64-bit values to the two inputs.
(composite)	<code>__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)</code>	Sets the 4 32-bit values to the 4 inputs.
(composite)	<code>__m128i _mm_set_epi16(short w7, short w6, short w5, short w4, short w3, short w2, short w1, short w0)</code>	Sets the 8 16-bit values to the 8 inputs.
(composite)	<code>__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)</code>	Sets the 16 8-bit values to the 16 inputs.
(composite)	<code>__m128i _mm_set1_epi64(__m64 q)</code>	Sets the 2 64-bit values to the input.
(composite)	<code>__m128i _mm_set1_epi32(int a)</code>	Sets the 4 32-bit values to the input.
(composite)	<code>__m128i _mm_set1_epi16(short a)</code>	Sets the 8 16-bit values to the input.
(composite)	<code>__m128i _mm_set1_epi8(char a)</code>	Sets the 16 8-bit values to the input.
(composite)	<code>__m128i _mm_setr_epi64(__m64 q1, __m64 q0)</code>	Sets the two 64-bit values to the two inputs in reverse order.
(composite)	<code>__m128i _mm_setr_epi32(int i3, int i2, int i1, int i0)</code>	Sets the 4 32-bit values to the 4 inputs in reverse order.
(composite)	<code>__m128i _mm_setr_epi16(short w7, short w6, short w5, short w4, short w3, short w2, short w1, short w0)</code>	Sets the 8 16-bit values to the 8 inputs in reverse order.
(composite)	<code>__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)</code>	Sets the 16 8-bit values to the 16 inputs in reverse order.
(composite)	<code>__m128i _mm_setzero_si128()</code>	Sets all bits to 0.
(composite)	<code>__m128 _mm_set_ps(float w)</code> <code>__m128 _mm_set1_ps(float w)</code>	Sets the four SP FP values to w.
(composite)	<code>__m128d _mm_set1_pd(double w)</code>	Sets the two DP FP values to w.
(composite)	<code>__m128d _mm_set_sd(double w)</code>	Sets the lower DP FP values to w.
(composite)	<code>__m128d _mm_set_pd(double z, double y)</code>	Sets the two DP FP values to the two inputs.
(composite)	<code>__m128 _mm_set_ps(float z, float y, float x, float w)</code>	Sets the four SP FP values to the four inputs.
(composite)	<code>__m128d _mm_setr_pd(double z, double y)</code>	Sets the two DP FP values to the two inputs in reverse order.
(composite)	<code>__m128 _mm_setr_ps(float z, float y, float x, float w)</code>	Sets the four SP FP values to the four inputs in reverse order.
(composite)	<code>__m128d _mm_setzero_pd(void)</code>	Clears the two DP FP values.
(composite)	<code>__m128 _mm_setzero_ps(void)</code>	Clears the four SP FP values.



**Table C-2. Composite Intrinsic**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
MOVSD + shuffle	<code>__m128d _mm_load_pd(double * p)</code> <code>__m128d _mm_load1_pd(double *p)</code>	Loads a single DP FP value, copying it into both DP FP values.
MOVSS + shuffle	<code>__m128 _mm_load_ps(float * p)</code> <code>__m128 _mm_load1_ps(float *p)</code>	Loads a single SP FP value, copying it into all four words.
MOVAPD + shuffle	<code>__m128d _mm_loadr_pd(double * p)</code>	Loads two DP FP values in reverse order. The address must be 16-byte-aligned.
MOVAPS + shuffle	<code>__m128 _mm_loadr_ps(float * p)</code>	Loads four SP FP values in reverse order. The address must be 16-byte-aligned.
MOVSD + shuffle	<code>void _mm_store1_pd(double *p, __m128d a)</code>	Stores the lower DP FP value across both DP FP values.
MOVSS + shuffle	<code>void _mm_store_ps(float * p, __m128 a)</code> <code>void _mm_store1_ps(float *p, __m128 a)</code>	Stores the lower SP FP value across four words.
MOVAPD + shuffle	<code>_mm_storer_pd(double * p, __m128d a)</code>	Stores two DP FP values in reverse order. The address must be 16-byte-aligned.
MOVAPS + shuffle	<code>_mm_storer_ps(float * p, __m128 a)</code>	Stores four SP FP values in reverse order. The address must be 16-byte-aligned.



intel®

# Index





## A

AAA instruction . . . . . 3-15  
 AAD instruction . . . . . 3-16  
 AAM instruction . . . . . 3-17  
 AAS instruction . . . . . 3-18  
 Abbreviations, opcode key . . . . . A-1  
 Access rights, segment descriptor . . . . . 3-366  
 ADC instruction . . . . . 3-19, 3-389  
 ADD instruction . . . 3-15, 3-19, 3-21, 3-174, 3-389  
 ADDPD instruction . . . . . 3-23  
 ADDPS instruction . . . . . 3-25  
 Addressing methods  
   codes . . . . . A-1  
   operand codes . . . . . A-3  
   register codes . . . . . A-3  
 Addressing, segments . . . . . 1-8  
 ADDSD instruction . . . . . 3-27  
 ADDSS instruction . . . . . 3-29  
 Advanced programmable interrupt controller (see APIC)  
 AND instruction . . . . . 3-31, 3-389  
 ANDNPD instruction . . . . . 3-37  
 ANDNPS instruction . . . . . 3-39  
 ANDPD instruction . . . . . 3-33  
 ANDPS instruction . . . . . 3-35  
 Arctangent, x87 FPU operation . . . . . 3-252  
 ARPL instruction . . . . . 3-41

## B

B (default stack size) flag, segment descriptor . . . . . 3-589, 3-650  
 Base (operand addressing) . . . . . 2-3  
 BCD integers  
   packed . . . . . 3-174, 3-176, 3-201, 3-203  
   unpacked . . . . . 3-15, 3-16, 3-17, 3-18  
 Binary numbers . . . . . 1-7  
 Binary-coded decimal (see BCD)  
 Bit order . . . . . 1-6  
 BOUND instruction . . . . . 3-43  
 BOUND range exceeded exception (#BR) . . . . . 3-43  
 Branch hints . . . . . 2-2  
 BSF instruction . . . . . 3-45  
 BSR instruction . . . . . 3-47  
 BSWAP instruction . . . . . 3-49  
 BT instruction . . . . . 3-50  
 BTC instruction . . . . . 3-52, 3-389  
 BTR instruction . . . . . 3-54, 3-389  
 BTS instruction . . . . . 3-56, 3-389  
 Byte order . . . . . 1-6

## C

Caches, invalidating (flushing) . . . . . 3-343, 3-780

Call gate . . . . . 3-362  
 CALL instruction . . . . . 3-58  
 Calls (see Procedure calls)  
 CBW instruction . . . . . 3-69  
 CDQ instruction . . . . . 3-172  
 CF (carry) flag, EFLAGS register . 3-19, 3-21, 3-50, 3-52, 3-54, 3-56, 3-71, 3-78, 3-177, 3-322, 3-326, 3-496, 3-661, 3-694, 3-705, 3-707, 3-728, 3-739  
 Classify floating-point value, x87 FPU operation . . . . . 3-299  
 CLC instruction . . . . . 3-71  
 CLD instruction . . . . . 3-72  
 CLFLUSH instruction . . . . . 3-73  
 CLI instruction . . . . . 3-75  
 CLTS instruction . . . . . 3-77  
 CMC instruction . . . . . 3-78  
 CMOVcc instructions . . . . . 3-79  
 CMP instruction . . . . . 3-83  
 CMPPD instruction . . . . . 3-85, 3-98  
 CMPPS instruction . . . . . 3-89  
 CMPS instruction . . . . . 3-93, 3-674  
 CMPSB instruction . . . . . 3-93  
 CMPSD instruction . . . . . 3-93, 3-96  
 CMPSS instruction . . . 3-100, 3-101, 3-102, 3-103  
 CMPSW instruction . . . . . 3-93  
 CMPXCHG instruction . . . . . 3-104, 3-389  
 CMPXCHG8B instruction . . . . . 3-106  
 COMISD instruction . . . . . 3-108  
 COMISS instruction . . . . . 3-111  
 Compatibility  
   software . . . . . 1-6  
 Compiler functional equivalents . . . . . 881  
 Compiler intrinsics . . . . . 881  
 Condition code flags, EFLAGS register . . . . . 3-79  
 Condition code flags, x87 FPU status word  
   flags affected by instructions . . . . . 3-11  
   setting . . . . . 3-293, 3-295, 3-299  
 Conditional jump . . . . . 3-354  
 Conforming code segment . . . . . 3-361, 3-366  
 Constants (floating point), loading . . . . . 3-242  
 Control registers, moving values to and from 3-437  
 Cosine, x87 FPU operation . . . . . 3-218, 3-272  
 CPL . . . . . 3-75, 3-777  
 CQPID instruction . . . . . 3-114  
   brand identification . . . . . 3-123  
   brand index . . . . . 3-117  
   cache and TLB characteristics . . . . . 3-115, 3-121  
   CLFLUSH instruction cache line size . . . 3-117  
   extended function CQPID information . . . 3-115  
   feature information . . . . . 3-118  
   initial APIC ID . . . . . 3-117  
   processor brand string . . . . . 3-115  
   processor type fields . . . . . 3-117

version information . . . . . 3-115, 3-116  
 CR0 control register . . . . . 3-718  
 CS register 3-58, 3-331, 3-346, 3-358, 3-432, 3-589  
 Current privilege level (see CPL)  
 CVTDQ2PD instruction . . . . . 3-128  
 CVTDQ2PS instruction . . . . . 3-130  
 CVTPD2DQ instruction . . . . . 3-132  
 CVTPD2PI instruction . . . . . 3-134  
 CVTPD2PS instruction . . . . . 3-136  
 CVTPI2PD instruction . . . . . 3-138  
 CVTPI2PS instruction . . . . . 3-140  
 CVTPS2DQ instruction . . . . . 3-142  
 CVTPS2PD instruction . . . . . 3-144  
 CVTPS2PI instruction . . . . . 3-146  
 CVTSD2SI instruction . . . . . 3-148  
 CVTSD2SS instruction . . . . . 3-150  
 CVTSI2SD instruction . . . . . 3-152  
 CVTSI2SS instruction . . . . . 3-154  
 CVTSS2SD instruction . . . . . 3-156  
 CVTSS2SI instruction . . . . . 3-158  
 CVTTPD2DQ instruction . . . . . 3-162  
 CVTTPD2PI instruction . . . . . 3-160  
 CVTTPS2DQ instruction . . . . . 3-164  
 CVTTPS2PI instruction . . . . . 3-166  
 CVTTSD2SI instruction . . . . . 3-168  
 CVTTSS2SI instruction . . . . . 3-170  
 CWD instruction . . . . . 3-172  
 CWDE instruction . . . . . 3-69  
 C/C++ compiler intrinsics  
   compiler functional equivalents . . . . . C-1  
   composite . . . . . C-28  
   descriptor of . . . . . 3-8  
   lists of . . . . . C-1  
   simple . . . . . C-3

**D**

D (default operation size) flag, segment descriptor  
 3-589, 3-593, 3-650  
 DAA instruction . . . . . 3-174  
 DAS instruction . . . . . 3-176  
 Debug registers, moving value to and from . 3-439  
 DEC instruction . . . . . 3-177, 3-389  
 Denormal number (see Denormalized finite  
 number)  
 Denormalized finite number . . . . . 3-299  
 DF (direction) flag, EFLAGS register . . 3-72, 3-94,  
 3-328, 3-391, 3-479, 3-519, 3-696, 3-729  
 Displacement (operand addressing) . . . . . 2-3  
 DIV instruction . . . . . 3-179  
 Divide error exception (#DE) . . . . . 3-179  
 DIVPD instruction . . . . . 3-182  
 DIVPS instruction . . . . . 3-184  
 DIVSD instruction . . . . . 3-186  
 DIVSS instruction . . . . . 3-188  
 DS register . . . . . 3-93, 3-371, 3-391, 3-479, 3-519

**E**

EDI register . . . . . 3-696, 3-729, 3-734  
 Effective address . . . . . 3-374  
 EFLAGS register  
   condition codes . . . . . 3-80, 3-210, 3-215  
   flags affected by instructions . . . . . 3-11  
   loading . . . . . 3-365  
   popping . . . . . 3-595  
   popping on return from interrupt . . . . . 3-346  
   pushing . . . . . 3-655  
   pushing on interrupts . . . . . 3-331  
   saving . . . . . 3-689  
   status flags . . . . . 3-83, 3-355, 3-699, 3-756  
 EIP register . . . . . 3-58, 3-331, 3-346, 3-358  
 EMMS instruction . . . . . 3-190  
 Encoding  
   cacheability and memory ordering instructions .  
     B-30  
   cacheability instructions . . . . . B-44  
   SIMD-integer register field . . . . . B-29, B-38  
 ENTER instruction . . . . . 3-191  
 ES register . . . . . 3-371, 3-519, 3-696, 3-734  
 ESI register . . . . . 3-93, 3-391, 3-479, 3-519, 3-729  
 ESP register . . . . . 3-59, 3-590  
 Exceptions  
   BOUND range exceeded (#BR) . . . . . 3-43  
   notation . . . . . 1-8  
   overflow exception (#OF) . . . . . 3-331  
   returning from . . . . . 3-346  
 Exponent, extracting from floating-point number . .  
 3-311  
 Extract exponent and significand, x87 FPU  
   operation . . . . . 3-311

**F**

F2XM1 instruction . . . . . 3-194, 3-311  
 FABS instruction . . . . . 3-196  
 FADD instruction . . . . . 3-198  
 FADDP instruction . . . . . 3-198  
 Far call, CALL instruction . . . . . 3-58  
 Far pointer, loading . . . . . 3-371  
 Far return, RET instruction . . . . . 3-677  
 FBLD instruction . . . . . 3-201  
 FBSTP instruction . . . . . 3-203  
 FCHS instruction . . . . . 3-206  
 FCLEX/FNCLEX instructions . . . . . 3-208  
 FCMOVCc instructions . . . . . 3-210  
 FCOM instruction . . . . . 3-212  
 FCOMI instruction . . . . . 3-215  
 FCOMIP instruction . . . . . 3-215  
 FCOMP instruction . . . . . 3-212  
 FCOMPP instruction . . . . . 3-212  
 FCOS instruction . . . . . 3-218  
 FDECSTP instruction . . . . . 3-220  
 FDIV instruction . . . . . 3-221  
 FDIVP instruction . . . . . 3-221  
 FDIVR instruction . . . . . 3-225

FDIVRP instruction	3-225
Feature information, processor	3-114, 3-120
FFREE instruction	3-229
FIADD instruction	3-198
FICOM instruction	3-230
FICOMP instruction	3-230
FIDIV instruction	3-221
FIDIVR instruction	3-225
FILD instruction	3-232
FIMUL instruction	3-248
FINCSTP instruction	3-234
FINIT/FNINIT instructions	3-235, 3-265
FIST instruction	3-237
FISTP instruction	3-237
FISUB instruction	3-287
FISUBR instruction	3-290
FLD instruction	3-240
FLD1 instruction	3-242
FLDCW instruction	3-244
FLDENV instruction	3-246
FLDL2E instruction	3-242
FLDL2T instruction	3-242
FLDLG2 instruction	3-242
FLDLN2 instruction	3-242
FLDPI instruction	3-242
FLDZ instruction	3-242
Floating-point exceptions	
SSE and SSE2 SIMD	3-13, 3-14
x87 FPU	3-13
Flushing	
caches	3-343, 3-780
TLB entry	3-345
FMUL instruction	3-248
FMULP instruction	3-248
FNOP instruction	3-251
FNSTENV instruction	3-246
FPATAN instruction	3-252
FPREM1 instruction	3-257
FPTAN instruction	3-260
FRNDINT instruction	3-262
FRSTOR instruction	3-263
FS register	3-371
FSAVE/FNSAVE instructions	3-263, 3-265
FSCALE instruction	3-268
FSIN instruction	3-270
FSINCOS instruction	3-272
FSQRT instruction	3-274
FST instruction	3-276
FSTCW/FNSTCW instructions	3-279
FSTENV/FNSTENV instructions	3-281
FSTP instruction	3-276
FSTSW/FNSTSW instructions	3-284
FSUB instruction	3-287
FSUBP instruction	3-287
FSUBR instruction	3-290
FSUBRP instruction	3-290
FTST instruction	3-293
FUCOM instruction	3-295

FUCOMI instruction	3-215
FUCOMIP instruction	3-215
FUCOMP instruction	3-295
FUCOMPP instruction	3-295
FXAM instruction	3-299
FXCH instruction	3-301
FXRSTOR instruction	3-303
FXSAVE instruction	3-305
FTRACT instruction	3-268, 3-311
FYL2X instruction	3-313
FYL2XP1 instruction	3-315

**G**

GDT (global descriptor table)	3-381, 3-384
GDTR (global descriptor table register)	3-381, 3-702
General-purpose registers	
moving value to and from	3-432
popping all	3-593
pushing all	3-653
GS register	3-371

**H**

Hexadecimal numbers	1-7
HLT instruction	3-317

**I**

IDIV instruction	3-318
IDT (interrupt descriptor table)	3-332, 3-381
IDTR (interrupt descriptor table register)	3-381, 3-702
IF (interrupt enable) flag, EFLAGS register	3-75, 3-730
Immediate operands	2-3
IMUL instruction	3-321
IN instruction	3-324
INC instruction	3-326, 3-389
Index (operand addressing)	2-3
Initialization x87 FPU	3-235
Input/output (see I/O)	
INS instruction	3-328, 3-674
INSB instruction	3-328
INSD instruction	3-328
Instruction format	
base field	2-3
description of reference information	3-1
displacement	2-3
illustration of	2-1
immediate	2-3
index field	2-3
Mod field	2-3
ModR/M byte	2-3
opcode	2-3
prefixes	2-1
reg/opcode field	2-3
r/m field	2-3

scale field . . . . . 2-3  
 SIB byte . . . . . 2-3  
 Instruction operands . . . . . 1-7  
 Instruction prefixes (see Prefixes)  
 Instruction reference, nomenclature . . . . . 3-1  
 Instruction set, reference . . . . . 3-1  
 INSW instruction . . . . . 3-328  
 INT 3 instruction . . . . . 3-331  
 Integer, storing, x87 FPU data type . . . . . 3-237  
 Inter-privilege level  
   call, CALL instruction . . . . . 3-58  
   return, RET instruction . . . . . 3-677  
 Interrupts  
   interrupt vector 4 . . . . . 3-331  
   returning from . . . . . 3-346  
   software . . . . . 3-331  
 INTn instruction . . . . . 3-331  
 INTO instruction . . . . . 3-331  
 Intrinsic . . . . . 881  
   compiler functional equivalents . . . . . C-1  
   composite . . . . . C-28  
   description of . . . . . 3-8  
   list of . . . . . C-1  
   simple . . . . . C-3  
 INVD instruction . . . . . 3-343  
 INVLPG instruction . . . . . 3-345  
 IOPL (I/O privilege level) field, EFLAGS register . . . . . 3-75, 3-655, 3-730  
 IRET instruction . . . . . 3-346  
 IRETD instruction . . . . . 3-346  
 I/O privilege level (see IOPL)

**J**

Jcc instructions . . . . . 3-354  
 JMP instruction . . . . . 3-358  
 Jump operation . . . . . 3-358

**L**

LAHF instruction . . . . . 3-365  
 LAR instruction . . . . . 3-366  
 LDMXCSR instruction . . . . . 3-369  
 LDS instruction . . . . . 3-371  
 LDT (local descriptor table) . . . . . 3-384  
 LDTR (local descriptor table register) 3-384, 3-716  
 LEA instruction . . . . . 3-374  
 LEAVE instruction . . . . . 3-376  
 LES instruction . . . . . 3-371  
 LFENCE instruction . . . . . 3-379  
 LFS instruction . . . . . 3-371  
 LGDT instruction . . . . . 3-381  
 LGS instruction . . . . . 3-371  
 LIDT instruction . . . . . 3-381  
 LLDT instruction . . . . . 3-384  
 LMSW instruction . . . . . 3-387  
 Load effective address operation . . . . . 3-374

LOCK prefix . . . . . 3-19, 3-21, 3-31, 3-52, 3-54, 3-56, 3-104, 3-106, 3-177, 3-326, 3-389, 3-506, 3-509, 3-511, 3-694, 3-739, 3-784, 3-786, 3-790  
 Locking operation . . . . . 3-389  
 LODS instruction . . . . . 3-391, 3-674  
 LODSB instruction . . . . . 3-391  
 LODSD instruction . . . . . 3-391  
 LODSW instruction . . . . . 3-391  
 Log epsilon, x87 FPU operation . . . . . 3-313  
 Log (base 2), x87 FPU operation . . . . . 3-315  
 LOOP instructions . . . . . 3-394  
 LOOPcc instructions . . . . . 3-394  
 LSL instruction . . . . . 3-396  
 LSS instruction . . . . . 3-371  
 LTR instruction . . . . . 3-400

**M**

Machine status word, CR0 register . . . . . 3-387, 3-718  
 MASKMOVDQU instruction . . . . . 3-402  
 MASKMOVQ instruction . . . . . 3-404  
 MAXPD instruction . . . . . 3-407  
 MAXPS instruction . . . . . 3-410  
 MAXSD instruction . . . . . 3-413  
 MAXSS instruction . . . . . 3-416  
 MFENCE instruction . . . . . 3-419  
 MINPD instruction . . . . . 3-420  
 MINPS . . . . . 3-423  
 MINPS instruction . . . . . 3-423  
 MINS instruction . . . . . 3-426  
 MINSS instruction . . . . . 3-429  
 Mod field, instruction format . . . . . 2-3  
 ModR/M byte  
   16-bit addressing forms . . . . . 2-5  
   32-bit addressing forms of . . . . . 2-6  
   description of . . . . . 2-3  
   format of . . . . . 2-1  
 MOV instruction . . . . . 3-432  
 MOV instruction (control registers) . . . . . 3-437  
 MOV instruction (debug registers) . . . . . 3-439  
 MOVAPD instruction . . . . . 3-441  
 MOVAPS instruction . . . . . 3-443  
 MOVD instruction . . . . . 3-445  
 MOVDDQ instruction . . . . . 3-451  
 MOVDDQA instruction . . . . . 3-447  
 MOVDDQU instruction . . . . . 3-449  
 MOVHPS instruction . . . . . 3-452  
 MOVHPD instruction . . . . . 3-453  
 MOVHPS instruction . . . . . 3-455  
 MOVHPS instruction . . . . . 3-455  
 MOVHPS instruction . . . . . 3-457  
 MOVLPD instruction . . . . . 3-458  
 MOVLPS instruction . . . . . 3-460  
 MOVMSKPD instruction . . . . . 3-462  
 MOVMSKPS instruction . . . . . 3-464  
 MOVNTDQ instruction . . . . . 3-466  
 MOVNTI instruction . . . . . 3-468  
 MOVNTPD instruction . . . . . 3-470



MOVNTPS instruction	3-472
MOVNTQ instruction	3-474
MOVQ instruction	3-476
MOVQ2DQ instruction	3-478
MOVS instruction	3-479, 3-674
MOVSB instruction	3-479
MOVSD instruction	3-479, 3-482
MOVSS instruction	3-485
MOVSW instruction	3-479
MOVX instruction	3-488
MOVUPD instruction	3-490
MOVUPS instruction	3-492
MOVZX instruction	3-494
MSRs (model specific registers)	
reading	3-669
writing	3-782
MUL instruction	3-17, 3-496
MULPD instruction	3-498
MULPS instruction	3-500
MULSD instruction	3-502
MULSS instruction	3-504

## N

NaN. testing for	3-293
Near	
call, CALL instruction	3-58
return, RET instruction	3-677
NEG instruction	3-389, 3-506
Nomenclature, used in instruction reference pages	3-1
Nonconforming code segment	3-362
NOP instruction	3-508
NOT instruction	3-389, 3-509
Notation	
bit and byte order	1-6
exceptions	1-8
hexadecimal and binary numbers	1-7
instruction operands	1-7
reserved bits	1-6
segmented addressing	1-8
Notational conventions	1-6
NT (nested task) flag, EFLAGS register	3-346

## O

OF (carry) flag, EFLAGS register	3-322
OF (overflow) flag, EFLAGS register	3-19, 3-21, 3-331, 3-496, 3-694, 3-705, 3-707, 3-739
Opcode	
escape instructions	A-12
map	A-1
Opcode extensions	
description	A-10
table	A-11
Opcode format	2-3
Opcode integer instructions	
one-byte	A-4

two-byte	A-6, A-7
two-byte opcode map	A-4
two-byte opcode map	A-8, A-9
Opcode key abbreviations	A-1
Operand, instruction	1-7
OR instruction	3-389, 3-511
ORPD instruction	3-513
ORPS instruction	3-515
OUT instruction	3-517
OUTS instruction	3-519, 3-674
OUTSB instruction	3-519
OUTSD instruction	3-519
OUTSW instruction	3-519
Overflow exception (#OF)	3-331
Overflow, FPU exception (see Numeric overflow exception)	

## P

PACKSSDW instruction	3-522, 3-523, 3-524, 3-525
PACKSSWB instruction	3-522, 3-523, 3-524, 3-525
PACKUSWB instruction	3-526, 3-527, 3-528
PADDQ instruction	3-532
PADDSB instruction	3-534
PADDSW instruction	3-534
PADDUSB instruction	3-537
PADDUSW instruction	3-537
PAND instruction	3-540
PANDN instruction	3-542
PAUSE instruction	3-544
PAVGB instruction	3-545
PAVGW instruction	3-545
PCMPEQB instruction	3-548
PCMPEQD instruction	3-548
PCMPEQW instruction	3-548
PCMPGTB instruction	3-552
PCMPGTD instruction	3-552
PCMPGTW instruction	3-552
PE (protection enable) flag, CR0 register	3-387
Performance-monitoring counters	
reading	3-671
PEXTRW instruction	3-556, 3-557
Pi	
loading	3-242
PINSRW instruction	3-558, 3-559, 3-560
PMADDWD instruction	3-561, 3-562, 3-563
PMAXSW instruction	3-564
PMAXUB instruction	3-567
PMINSW instruction	3-570, 3-571, 3-572
PMINUB instruction	3-573
PMOVMASKB instruction	3-576
PMULHUW instruction	3-578
PMULHW instruction	3-581
PMULLW instruction	3-584
PMULUDQ instruction	3-587
POP instruction	3-589
POPA instruction	3-593
POPAD instruction	3-593

- POPF instruction . . . . . 3-595  
 POPFD instruction . . . . . 3-595  
 POR instruction . . . . . 3-598  
 PREFETCHH instruction . . . . . 3-600  
 Prefixes  
   branch hints . . . . . 2-2  
   instruction, description of . . . . . 2-1  
   LOCK . . . . . 3-389  
   REP/REPE/REPZ/REPNE/REPZ . . . . . 3-674  
 PSADBW instruction . . . . . 3-602  
 PSHUFD instruction . . . . . 3-605  
 PSHUFW instruction . . . . . 3-608  
 PSHUFW instruction . . . . . 3-612  
 PSLLD instruction . . . . . 3-615  
 PSLLDQ instruction . . . . . 3-614  
 PSLLQ instruction . . . . . 3-615  
 PSLLW instruction . . . . . 3-615  
 PSRAD instruction . . . . . 3-620  
 PSRAW instruction . . . . . 3-620  
 PSRLD instruction . . . . . 3-625  
 PSRLDQ instruction . . . . . 3-624  
 PSRLQ instruction . . . . . 3-625  
 PSRLW instruction . . . . . 3-625  
 PSUBB instruction . . . . . 3-630  
 PSUBD instruction . . . . . 3-630  
 PSUBQ instruction . . . . . 3-634  
 PSUBSB instruction . . . . . 3-636  
 PSUBSW instruction . . . . . 3-636  
 PSUBUSB instruction . . . . . 3-639  
 PSUBUSW instruction . . . . . 3-639  
 PSUBW instruction . . . . . 3-630  
 PUNPCKHBW instruction . . . . . 3-642, 3-643, 3-644,  
   3-645  
 PUNPCKHDQ instruction . . . . . 3-642, 3-643, 3-644,  
   3-645  
 PUNPCKHWD instruction . . . . . 3-642, 3-643, 3-644,  
   3-645  
 PUNPCKLBW instruction . . . . . 3-646  
 PUNPCKLDQ instruction . . . . . 3-646  
 PUNPCKLWD instruction . . . . . 3-646  
 PUSH instruction . . . . . 3-650  
 PUSHA instruction . . . . . 3-653  
 PUSHAD instruction . . . . . 3-653  
 PUSHF instruction . . . . . 3-655  
 PUSHFD instruction . . . . . 3-655  
 PXOR instruction . . . . . 3-657
- Q**  
 Quiet NaN (see QNaN)
- R**  
 RC (rounding control) field, x87 FPU control word.  
   3-238, 3-242, 3-276  
 RCL instruction . . . . . 3-660  
 RCPPS instruction . . . . . 3-665  
 RCPSS instruction . . . . . 3-667  
 RCR instruction . . . . . 3-660  
 RDMSR instruction . . . . . 3-669, 3-673  
 RDPMS instruction . . . . . 3-671  
 RDTSC instruction . . . . . 3-673  
 Reg/opcode field, instruction format . . . . . 2-3  
 Related literature . . . . . 1-9  
 Remainder, x87 FPU operation . . . . . 3-257  
 REP/REPE/REPZ/REPNE/REPZ prefixes . . . . . 3-94,  
   3-329, 3-520, 3-674  
 Reserved bits . . . . . 1-6  
 RET instruction . . . . . 3-677  
 ROL instruction . . . . . 3-660  
 ROR instruction . . . . . 3-660  
 Rotate operation . . . . . 3-660  
 Rounding, round to integer, x87 FPU operation . . . . .  
   3-262  
 RPL field . . . . . 3-41  
 RSM instruction . . . . . 3-684  
 RSQRTPS instruction . . . . . 3-685  
 RSQRTSS instruction . . . . . 3-687  
 R/m field, instruction format . . . . . 2-3
- S**  
 SAL instruction . . . . . 3-690  
 SAR instruction . . . . . 3-690  
 SBB instruction . . . . . 3-389, 3-694  
 Scale (operand addressing) . . . . . 2-3  
 Scale, x87 FPU operation . . . . . 3-268  
 SCAS instruction . . . . . 3-674, 3-696  
 SCASB instruction . . . . . 3-696  
 SCASD instruction . . . . . 3-696  
 SCASW instruction . . . . . 3-696  
 Segment  
   descriptor, segment limit . . . . . 3-396  
   limit . . . . . 3-396  
   registers, moving values to and from . . . . . 3-432  
   selector, RPL field . . . . . 3-41  
 Segmented addressing . . . . . 1-8  
 SETcc instructions . . . . . 3-699  
 SF (sign) flag, EFLAGS register . . . . . 3-19, 3-21  
 SFENCE instruction . . . . . 3-701  
 SGDT instruction . . . . . 3-702  
 SHAF instruction . . . . . 3-689  
 SHL instruction . . . . . 3-690  
 SHLD instruction . . . . . 3-705  
 SHR instruction . . . . . 3-690  
 SHRD instruction . . . . . 3-707  
 SHUFPD instruction . . . . . 3-709  
 SHUFPS instruction . . . . . 3-712  
 SIB byte  
   32-bit addressing forms of . . . . . 2-7  
   description of . . . . . 2-3  
   format of . . . . . 2-1  
 SIDT instruction . . . . . 3-702  
 Signaling NaN (see SNaN)  
 Significant, extracting from floating-point number . . . . .  
   3-311

SIMD floating-point exceptions, unmasking, effects of . . . . . 3-369

Sine, x87 FPU operation . . . . . 3-270, 3-272

SLDT instruction . . . . . 3-716

SMSW instruction . . . . . 3-718

SQRTPD instruction . . . . . 3-720

SQRTPS instruction . . . . . 3-722

SQRTSD instruction . . . . . 3-724

SQRTSS instruction . . . . . 3-726

Square root, Fx87 PU operation . . . . . 3-274

SS register . . . . . 3-371, 3-433, 3-590

SSE extensions

- encoding cacheability and memory ordering instructions . . . . . B-30
- encoding SIMD-integer register field . . . . . B-29

SSE2 extensions

- encoding cacheability instructions . . . . . B-44
- encoding SIMD-integer register field . . . . . B-38

Stack (see Procedure stack)

Stack, pushing values on . . . . . 3-650

Status flags, EFLAGS register . . . 3-80, 3-83, 3-210, 3-215, 3-355, 3-699, 3-756

STC instruction . . . . . 3-728

STD instruction . . . . . 3-729

STI instruction . . . . . 3-730

STMXCSR instruction . . . . . 3-732

STOS instruction . . . . . 3-674, 3-734

STOSB instruction . . . . . 3-734

STOSD instruction . . . . . 3-734

STOSW instruction . . . . . 3-734

STR instruction . . . . . 3-737

String instructions 3-93, 3-328, 3-391, 3-479, 3-519, 3-696, 3-734

SUB instruction . . . . . 3-18, 3-176, 3-389, 3-739

SUBPD instruction . . . . . 3-741

SUBSS instruction . . . . . 3-747

SYSENTER instruction . . . . . 3-749

SYSEXIT instruction . . . . . 3-753

**T**

Tangent, x87 FPU operation . . . . . 3-260

Task gate . . . . . 3-362

Task register

- loading . . . . . 3-400
- storing . . . . . 3-737

Task state segment (see TSS)

Task switch

- CALL instruction . . . . . 3-58
- return from nested task, IRET instruction . 3-346

TEST instruction . . . . . 3-756

Time-stamp counter, reading . . . . . 3-673

TLB entry, invalidating (flushing) . . . . . 3-345

TS (task switched) flag, CR0 register . . . . . 3-77

TSD flag, CR4 register . . . . . 3-673

TSS, relationship to task register . . . . . 3-737

**U**

UCOMISD instruction . . . . . 3-758

UCOMISS instruction . . . . . 3-761

UD2 instruction . . . . . 3-764

Undefined, format opcodes . . . . . 3-293

Underflow, FPU exception (see Numeric underflow exception)

Unordered values . . . . . 3-212, 3-215, 3-293, 3-295

UNPCKHPD instruction . . . . . 3-765

UNPCKHPS instruction . . . . . 3-768

UNPCKLPD instruction . . . . . 3-771

UNPCKLPS instruction . . . . . 3-774

**V**

Vector (see Interrupt vector)

VERR instruction . . . . . 3-777

Version information, processor . . . . . 3-114, 3-120

VERW instruction . . . . . 3-777

VM (virtual 8086 mode) flag, EFLAGS register . . . 3-346

**W**

WAIT/FWAIT instructions . . . . . 3-779

WBINVD instruction . . . . . 3-780

Write-back and invalidate caches . . . . . 3-780

WRMSR instruction . . . . . 3-782

**X**

x87 FPU

- checking for pending x87 FPU exceptions 3-779
- constants . . . . . 3-242
- initialization . . . . . 3-235

x87 FPU control word

- loading . . . . . 3-244, 3-246
- RC field . . . . . 3-238, 3-242, 3-276
- restoring . . . . . 3-263
- saving . . . . . 3-265, 3-281
- storing . . . . . 3-279

x87 FPU data pointer . . 3-246, 3-263, 3-265, 3-281

x87 FPU instruction pointer . 3-246, 3-263, 3-265, 3-281

x87 FPU last opcode . . 3-246, 3-263, 3-265, 3-281

x87 FPU status word

- condition code flags 3-212, 3-230, 3-293, 3-295, 3-299
- loading . . . . . 3-246
- restoring . . . . . 3-263
- saving . . . . . 3-265, 3-281, 3-284
- TOP field . . . . . 3-234
- x87 FPU flags affected by instructions . . . 3-11

x87 FPU tag word . . . . 3-246, 3-263, 3-265, 3-281

XADD instruction . . . . . 3-389, 3-784

XCHG instruction . . . . . 3-389, 3-786

XLAT/XLATB instruction . . . . . 3-788

XOR instruction . . . . . 3-389, 3-790



XORPD instruction . . . . .3-792  
XORPS instruction . . . . .3-794

**Z**

ZF (zero) flag, EFLAGS register . . . 3-104, 3-106,  
3-366, 3-394, 3-396, 3-674, 3-777

