IA-32 Intel® Architecture Software Developer's Manual

Volume 2A: Instruction Set Reference, A-M

NOTE: The *IA-32 Intel Architecture Software Developer's Manual* consists of four volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-M,* Order Number 253666; *Instruction Set Reference N-Z,* Order Number 253667; and the *System Programming Guide,* Order Number 253668. Refer to all four volumes when evaluating your design needs.

2004

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

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

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

The Intel® IA-32 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.

Hyper-Threading Technology requires a computer system with an Intel® Pentium® 4 processor supporting Hyper-Threading Technology and an HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See http://www.intel.com/info/hyperthreading/ for more information including details on which processors support HT Technology.

Intel, Intel386, Intel486, Pentium, Intel Xeon, Intel NetBurst, Intel SpeedStep, OverDrive, MMX, Celeron, and Itanium are trademarks or registered trademarks of Intel Corporation and its subsidiaries in the United States and other countries.

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

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

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

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

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

Copyright © 1997 - 2004 Intel Corporation

CONTENTS FOR VOLUME 2A AND 2B

PAGE

CHAPTER 1
ABOUT THIS MANUAL
1.1. IA-32 PROCESSORS COVERED IN THIS MANUAL 1-1
1.2. OVERVIEW OF THE *IA-32 INTEL® ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUMES 2A & 2B*: INSTRUCTION SET REFERENCE 1-2
1.3. NOTATIONAL CONVENTIONS 1-2
1.3.1. Bit and Byte Order 1-2
1.3.2. Reserved Bits and Software Compatibility 1-3
1.3.3. Instruction Operands 1-4
1.3.4. Hexadecimal and Binary Numbers 1-4
1.3.5. Segmented Addressing 1-5
1.3.6. Exceptions 1-5
1.4. RELATED LITERATURE 1-6

CHAPTER 2
INSTRUCTION FORMAT
2.1. GENERAL INSTRUCTION FORMAT 2-1
2.2. SUMMARY OF INSTRUCTION PREFIXES 2-2
2.3. OPCODES 2-3
2.4. MODR/M AND SIB BYTES 2-4
2.5. DISPLACEMENT AND IMMEDIATE BYTES 2-4
2.6. ADDRESSING-MODE ENCODING OF MODR/M AND SIB BYTES 2-5

CHAPTER 3
INSTRUCTION SET REFERENCE, A-M
3.1. INTERPRETING THE INSTRUCTION REFERENCE PAGES 3-1
3.1.1. Instruction Format 3-1
3.1.1.1. Opcode Column 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 Intrinsics Equivalents 3-9
3.1.3.1. The Intrinsics API 3-9
3.1.3.2. MMX™ Technology Intrinsics 3-9
3.1.3.3. SSE/SSE2/SSE3 Intrinsics 3-10
3.1.4. Flags Affected 3-11
3.1.5. FPU Flags Affected 3-12
3.1.6. Protected Mode Exceptions 3-12
3.1.7. Real-Address Mode Exceptions 3-14
3.1.8. Virtual-8086 Mode Exceptions 3-14
3.1.9. Floating-Point Exceptions 3-14
3.1.10. SIMD Floating-Point Exceptions 3-14
3.2. INSTRUCTION REFERENCE 3-15
AAA—ASCII Adjust After Addition 3-16
AAD—ASCII Adjust AX Before Division 3-17
AAM—ASCII Adjust AX After Multiply 3-18

PAGE

AAS—ASCII Adjust AL After Subtraction 3-19
ADC—Add with Carry 3-20
ADD—Add 3-22
ADDPD—Add Packed Double-Precision Floating-Point Values 3-24
ADDPS—Add Packed Single-Precision Floating-Point Values 3-26
ADDSD—Add Scalar Double-Precision Floating-Point Values 3-28
ADDSS—Add Scalar Single-Precision Floating-Point Values 3-30
ADDSUBPD: Packed Double-FP Add/Subtract 3-32
ADDSUBPS: Packed Single-FP Add/Subtract 3-35
AND—Logical AND 3-38
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values 3-40
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values 3-42
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values 3-44
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values 3-46
ARPL—Adjust RPL Field of Segment Selector 3-48
BOUND—Check Array Index Against Bounds 3-50
BSF—Bit Scan Forward 3-52
BSR—Bit Scan Reverse 3-54
BSWAP—Byte Swap 3-56
BT—Bit Test 3-57
BTC—Bit Test and Complement 3-59
BTR—Bit Test and Reset 3-61
BTS—Bit Test and Set 3-63
CALL—Call Procedure 3-65
CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword 3-76
CDQ—Convert Double to Quad 3-77
CLC—Clear Carry Flag 3-78
CLD—Clear Direction Flag 3-79
CLFLUSH—Flush Cache Line 3-80
CLI — Clear Interrupt Flag 3-82
CLTS—Clear Task-Switched Flag in CR0 3-84
CMC—Complement Carry Flag 3-85
CMOVcc—Conditional Move 3-86
CMP—Compare Two Operands 3-89
CMPPD—Compare Packed Double-Precision Floating-Point Values 3-91
CMPPS—Compare Packed Single-Precision Floating-Point Values 3-95
CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands 3-99
CMPSD—Compare Scalar Double-Precision Floating-Point Values 3-102
CMPSS—Compare Scalar Single-Precision Floating-Point Values 3-106
CMPXCHG—Compare and Exchange 3-110
CMPXCHG8B—Compare and Exchange 8 Bytes 3-112
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS 3-114

PAGE

COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS 3-117
CPUID—CPU Identification 3-120
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values 3-142
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values 3-144
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers 3-146
CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers 3-148
CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values 3-150
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values 3-152
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values 3-154
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers 3-156
CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values 3-158
CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers 3-160
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer 3-162
CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value 3-164
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value 3-166
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value 3-168
CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value 3-170
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer 3-172
CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers 3-174
CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers 3-176
CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers 3-178
CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers 3-180
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer 3-182
CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer 3-184

CWD/CDQ—Convert Word to Doubleword/Convert Doubleword to Quadword 3-186
CWDE—Convert Word to Doubleword 3-187
DAA—Decimal Adjust AL after Addition 3-188
DAS—Decimal Adjust AL after Subtraction 3-190
DEC—Decrement by 1 3-192
DIV—Unsigned Divide 3-194
DIVPD—Divide Packed Double-Precision Floating-Point Values 3-197
DIVPS—Divide Packed Single-Precision Floating-Point Values 3-199
DIVSD—Divide Scalar Double-Precision Floating-Point Values 3-201
DIVSS—Divide Scalar Single-Precision Floating-Point Values 3-203
EMMS—Empty MMX Technology State 3-205
ENTER—Make Stack Frame for Procedure Parameters 3-206
F2XM1—Compute 2x–1 3-209
FABS—Absolute Value 3-211
FADD/FADDP/FIADD—Add 3-212
FBLD—Load Binary Coded Decimal 3-215
FBSTP—Store BCD Integer and Pop 3-217
FCHS—Change Sign 3-220
FCLEX/FNCLEX—Clear Exceptions 3-221
FCMOVcc—Floating-Point Conditional Move 3-223
FCOM/FCOMP/FCOMPP—Compare Floating Point Values 3-225
FCOMI/FCOMIP/ FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS 3-228
FCOS—Cosine 3-231
FDECSTP—Decrement Stack-Top Pointer 3-233
FDIV/FDIVP/FIDIV—Divide 3-234
FDIVR/FDIVRP/FIDIVR—Reverse Divide 3-237
FFREE—Free Floating-Point Register 3-240
FICOM/FICOMP—Compare Integer 3-241
FILD—Load Integer 3-243
FINCSTP—Increment Stack-Top Pointer 3-245
FINIT/FNINIT—Initialize Floating-Point Unit 3-246
FIST/FISTP—Store Integer 3-248
FISTTP: Store Integer with Truncation 3-251
FLD—Load Floating Point Value 3-253
FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant 3-255
FLDCW—Load x87 FPU Control Word 3-257
FLDENV—Load x87 FPU Environment 3-259
FMUL/FMULP/FIMUL—Multiply 3-261
FNOP—No Operation 3-264
FPATAN—Partial Arctangent 3-265
FPREM—Partial Remainder 3-267
FPREM1—Partial Remainder 3-270
FPTAN—Partial Tangent 3-273
FRNDINT—Round to Integer 3-275
FRSTOR—Restore x87 FPU State 3-276

PAGE

FSAVE/FNSAVE—Store x87 FPU State 3-278
FSCALE—Scale 3-281
FSIN—Sine 3-283
FSINCOS—Sine and Cosine 3-285
FSQRT—Square Root 3-287
FST/FSTP—Store Floating Point Value 3-289
FSTCW/FNSTCW—Store x87 FPU Control Word 3-292
FSTENV/FNSTENV—Store x87 FPU Environment 3-294
FSTSW/FNSTSW—Store x87 FPU Status Word 3-297
FSUB/FSUBP/FISUB—Subtract 3-299
FSUBR/FSUBRP/FISUBR—Reverse Subtract 3-302
FTST—TEST 3-305
FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values 3-307
FWAIT—Wait 3-310
FXAM—Examine 3-311
FXCH—Exchange Register Contents 3-313
FXRSTOR—Restore x87 FPU, MMX Technology, SSE, and SSE2 State . . . 3-315
FXSAVE—Save x87 FPU, MMX Technology, SSE, and SSE2 State 3-318
FXTRACT—Extract Exponent and Significand 3-325
FYL2X—Compute y * log2x 3-327
FYL2XP1—Compute y * log2(x +1) 3-329
HADDPD: Packed Double-FP Horizontal Add 3-331
HADDPS: Packed Single-FP Horizontal Add 3-334
HLT—Halt 3-338
HSUBPD: Packed Double-FP Horizontal Subtract 3-339
HSUBPS: Packed Single-FP Horizontal Subtract 3-342
IDIV—Signed Divide 3-346
IMUL—Signed Multiply 3-349
IN—Input from Port 3-352
INC—Increment by 1 3-354
INS/INSB/INSW/INSD—Input from Port to String 3-356
INT n/INTO/INT 3—Call to Interrupt Procedure 3-359
INVD—Invalidate Internal Caches 3-370
INVLPG—Invalidate TLB Entry 3-372
IRET/IRETD—Interrupt Return 3-373
Jcc—Jump if Condition Is Met 3-380
JMP—Jump 3-384
LAHF—Load Status Flags into AH Register 3-391
LAR—Load Access Rights Byte 3-392
LDDQU: Load Unaligned Integer 128 Bits 3-395
LDMXCSR—Load MXCSR Register 3-397
LDS/LES/LFS/LGS/LSS—Load Far Pointer 3-399
LEA—Load Effective Address 3-402
LEAVE—High Level Procedure Exit 3-404
LES—Load Far Pointer 3-406
LFENCE—Load Fence 3-407

PAGE

LFS—Load Far Pointer 3-408
LGDT/LIDT—Load Global/Interrupt Descriptor Table Register 3-409
LLDT—Load Local Descriptor Table Register 3-411
LIDT—Load Interrupt Descriptor Table Register 3-413
LMSW—Load Machine Status Word 3-414
LOCK—Assert LOCK# Signal Prefix 3-416
LODS/LODSB/LODSW/LODSD—Load String 3-418
LOOP/LOOPcc—Loop According to ECX Counter 3-421
LSL—Load Segment Limit 3-423
LSS—Load Far Pointer 3-426
LTR—Load Task Register 3-427
MASKMOVDQU—Store Selected Bytes of Double Quadword 3-429
MASKMOVQ—Store Selected Bytes of Quadword 3-431
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values 3-434
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values 3-437
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value 3-440
MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value 3-442
MFENCE—Memory Fence 3-444
MINPD—Return Minimum Packed Double-Precision Floating-Point Values . . 3-445
MINPS—Return Minimum Packed Single-Precision Floating-Point Values . . . 3-448
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value 3-451
MINSS—Return Minimum Scalar Single-Precision Floating-Point Value 3-453
MONITOR: Setup Monitor Address 3-455
MOV—Move 3-458
MOV—Move to/from Control Registers 3-462
MOV—Move to/from Debug Registers 3-464
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values 3-466
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values 3-468
MOVD—Move Doubleword 3-470
MOVDDUP: Move One Double-FP and Duplicate 3-473
MOVDQA—Move Aligned Double Quadword 3-476
MOVDQU—Move Unaligned Double Quadword 3-478
MOVDQ2Q—Move Quadword from XMM to MMX Technology Register 3-480
MOVHLPS— Move Packed Single-Precision Floating-Point Values High to Low 3-481
MOVHPD—Move High Packed Double-Precision Floating-Point Value 3-482
MOVHPS—Move High Packed Single-Precision Floating-Point Values 3-484
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High 3-486
MOVLPD—Move Low Packed Double-Precision Floating-Point Value 3-487
MOVLPS—Move Low Packed Single-Precision Floating-Point Values 3-489
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask . . 3-491

PAGE

MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask . . 3-492
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint 3-493
MOVNTI—Store Doubleword Using Non-Temporal Hint 3-495
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint 3-497
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint 3-499
MOVNTQ—Store of Quadword Using Non-Temporal Hint 3-501
MOVSHDUP: Move Packed Single-FP High and Duplicate 3-503
MOVSLDUP: Move Packed Single-FP Low and Duplicate 3-506
MOVQ—Move Quadword 3-509
MOVQ2DQ—Move Quadword from MMX Technology to XMM Register 3-511
MOVS/MOVSB/MOVSW/MOVSD—Move Data from String to String 3-512
MOVSD—Move Scalar Double-Precision Floating-Point Value 3-515
MOVSS—Move Scalar Single-Precision Floating-Point Values 3-517
MOVSX—Move with Sign-Extension 3-519
MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values 3-520
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values 3-522
MOVZX—Move with Zero-Extend 3-524
MUL—Unsigned Multiply 3-525
MULPD—Multiply Packed Double-Precision Floating-Point Values 3-527
MULPS—Multiply Packed Single-Precision Floating-Point Values 3-529
MULSD—Multiply Scalar Double-Precision Floating-Point Values 3-531
MULSS—Multiply Scalar Single-Precision Floating-Point Values 3-533
MWAIT: Monitor Wait 3-535

CHAPTER 4
INSTRUCTION SET REFERENCE, N-Z

NEG—Two's Complement Negation 4-1
NOP—No Operation 4-3
NOT—One's Complement Negation 4-4
OR—Logical Inclusive OR 4-6
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values 4-8
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values 4-10
OUT—Output to Port 4-12
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port 4-14
PACKSSWB/PACKSSDW—Pack with Signed Saturation 4-17
PACKUSWB—Pack with Unsigned Saturation 4-21
PADDB/PADDW/PADDD—Add Packed Integers 4-24
PADDQ—Add Packed Quadword Integers 4-27
PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation . . . 4-29
PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation 4-32
PAND—Logical AND 4-35
PANDN—Logical AND NOT 4-37

PAGE

PAUSE—Spin Loop Hint 4-39
PAVGB/PAVGW—Average Packed Integers 4-40
PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal 4-43
PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than 4-47
PEXTRW—Extract Word 4-51
PINSRW—Insert Word 4-53
PMADDWD—Multiply and Add Packed Integers 4-55
PMAXSW—Maximum of Packed Signed Word Integers 4-58
PMAXUB—Maximum of Packed Unsigned Byte Integers 4-61
PMINSW—Minimum of Packed Signed Word Integers 4-64
PMINUB—Minimum of Packed Unsigned Byte Integers 4-67
PMOVMSKB—Move Byte Mask 4-70
PMULHUW—Multiply Packed Unsigned Integers and Store High Result 4-72
PMULHW—Multiply Packed Signed Integers and Store High Result 4-75
PMULLW—Multiply Packed Signed Integers and Store Low Result 4-78
PMULUDQ—Multiply Packed Unsigned Doubleword Integers 4-81
POP—Pop a Value from the Stack 4-83
POPA/POPAD—Pop All General-Purpose Registers 4-88
POPF/POPFD—Pop Stack into EFLAGS Register 4-90
POR—Bitwise Logical OR 4-93
PREFETCHh—Prefetch Data Into Caches 4-95
PSADBW—Compute Sum of Absolute Differences 4-97
PSHUFD—Shuffle Packed Doublewords 4-100
PSHUFHW—Shuffle Packed High Words 4-102
PSHUFLW—Shuffle Packed Low Words 4-104
PSHUFW—Shuffle Packed Words 4-106
PSLLDQ—Shift Double Quadword Left Logical 4-108
PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical 4-109
PSRAW/PSRAD—Shift Packed Data Right Arithmetic 4-113
PSRLDQ—Shift Double Quadword Right Logical 4-117
PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical 4-118
PSUBB/PSUBW/PSUBD—Subtract Packed Integers 4-122
PSUBQ—Subtract Packed Quadword Integers 4-125
PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation 4-127
PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation 4-130
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ—Unpack High Data 4-133
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data 4-137
PUSH—Push Word or Doubleword Onto the Stack 4-141
PUSHA/PUSHAD—Push All General-Purpose Registers 4-144
PUSHF/PUSHFD—Push EFLAGS Register onto the Stack 4-146
PXOR—Logical Exclusive OR 4-148
RCL/RCR/ROL/ROR—Rotate 4-150

PAGE

RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values 4-154
RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values 4-156
RDMSR—Read from Model Specific Register. 4-158
RDPMC—Read Performance-Monitoring Counters 4-159
RDTSC—Read Time-Stamp Counter 4-162
REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix 4-164
RET—Return from Procedure 4-167
ROL/ROR—Rotate 4-173
RSM—Resume from System Management Mode. 4-174
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values 4-175
RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value 4-177
SAHF—Store AH into Flags. 4-179
SAL/SAR/SHL/SHR—Shift 4-180
SBB—Integer Subtraction with Borrow 4-184
SCAS/SCASB/SCASW/SCASD—Scan String 4-186
SETcc—Set Byte on Condition 4-189
SFENCE—Store Fence 4-191
SGDT—Store Global Descriptor Table Register 4-192
SHL/SHR—Shift Instructions 4-194
SHLD—Double Precision Shift Left 4-195
SHRD—Double Precision Shift Right 4-197
SHUFPD—Shuffle Packed Double-Precision Floating-Point Values. 4-199
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values 4-201
SIDT—Store Interrupt Descriptor Table Register 4-204
SLDT—Store Local Descriptor Table Register 4-206
SMSW—Store Machine Status Word 4-208
SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values 4-210
SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values 4-212
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value. 4-214
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value. 4-216
STC—Set Carry Flag 4-218
STD—Set Direction Flag 4-219
STI—Set Interrupt Flag 4-220
STMXCSR—Store MXCSR Register State 4-223
STOS/STOSB/STOSW/STOSD—Store String 4-225
STR—Store Task Register 4-228
SUB—Subtract. 4-229
SUBPD—Subtract Packed Double-Precision Floating-Point Values 4-231
SUBPS—Subtract Packed Single-Precision Floating-Point Values. 4-233

PAGE

SUBSD—Subtract Scalar Double-Precision Floating-Point Values 4-235
SUBSS—Subtract Scalar Single-Precision Floating-Point Values 4-237
SYSENTER—Fast System Call 4-239
SYSEXIT—Fast Return from Fast System Call. 4-243
TEST—Logical Compare. 4-246
UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS 4-248
UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS 4-251
UD2—Undefined Instruction 4-254
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values. 4-255
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values. 4-257
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values. 4-259
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values 4-261
VERR, VERW—Verify a Segment for Reading or Writing. 4-263
WAIT/FWAIT—Wait. 4-265
WBINVD—Write Back and Invalidate Cache 4-266
WRMSR—Write to Model Specific Register 4-268
XADD—Exchange and Add. 4-270
XCHG—Exchange Register/Memory with Register 4-272
XLAT/XLATB—Table Look-up Translation 4-274
XOR—Logical Exclusive OR 4-276
XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values 4-278
XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values 4-280

APPENDIX A
OPCODE MAP
A.1. NOTES ON USING OPCODE TABLES. A-1
A.2. KEY TO ABBREVIATIONS A-1
A.2.1. Codes for Addressing Method. A-2
A.2.2. Codes for Operand Type. A-3
A.2.3. Register Codes A-3
A.3. OPCODE LOOK-UP EXAMPLES A-4
A.3.1. One-Byte Opcode Instructions A-4
A.3.2. Two-Byte Opcode Instructions A-5
A.3.3. Opcode Map Notes A-6
A.3.4. Opcode Extensions For One- And Two-byte Opcodes A-13
A.3.5. Escape Opcode Instructions A-14
A.3.5.1. Opcodes with ModR/M Bytes in the 00H through BFH Range. A-15
A.3.5.2. Opcodes with ModR/M Bytes outside the 00H through BFH Range A-15
A.3.5.3. Escape Opcodes with D8 as First Byte A-15
A.3.5.4. Escape Opcodes with D9 as First Byte A-17
A.3.5.5. Escape Opcodes with DA as First Byte A-18

PAGE

A.3.5.6. Escape Opcodes with DB as First Byte A-19
A.3.5.7. Escape Opcodes with DC as First Byte A-20
A.3.5.8. Escape Opcodes with DD as First Byte A-21
A.3.5.9. Escape Opcodes with DE as First Byte A-22
A.3.5.10. Escape Opcodes with DF As First Byte A-23

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 (tttn) B-5
B.1.7. Direction (d) Bit B-5
B.1.8. Other Notes B-6
B.2. GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS B-6
B.3. PENTIUM FAMILY INSTRUCTION FORMATS AND ENCODINGS B-19
B.4. MMX INSTRUCTION FORMATS AND ENCODINGS B-20
B.4.1. Granularity Field (gg) B-20
B.4.2. MMX Technology and General-Purpose Register Fields (mmxreg and reg) B-20
B.4.3. MMX Instruction Formats and Encodings Table B-20
B.5. P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS B-24
B.6. SSE INSTRUCTION FORMATS AND ENCODINGS B-25
B.7. SSE2 INSTRUCTION FORMATS AND ENCODINGS B-33
B.7.1. Granularity Field (gg) B-33
B.7.2. SSE3 Formats and Encodings Table B-46
B.8. FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS B-48

APPENDIX C
INTEL® C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS
C.1. SIMPLE INTRINSICS C-3
C.2. COMPOSITE INTRINSICS C-32

FIGURES

PAGE

Figure 1-1. Bit and Byte Order 1-3
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 3-8
Figure 3-3. ADDSUBPD: Packed Double-FP Add/Subtract 3-32
Figure 3-4. ADDSUBPS: Packed Single-FP Add/Subtract 3-35
Figure 3-5. Version Information Returned by CPUID in EAX 3-125
Figure 3-6. Extended Feature Information Returned in the ECX Register 3-127
Figure 3-7. Feature Information Returned in the EDX Register 3-128
Figure 3-8. Determination of Support for the Processor Brand String 3-135
Figure 3-9. Algorithm for Extracting Maximum Processor Frequency. 3-137
Figure 3-10. HADDPD: Packed Double-FP Horizontal Add 3-331
Figure 3-11. HADDPS: Packed Single-FP Horizontal Add 3-335
Figure 3-12. HSUBPD: Packed Double-FP Horizontal Subtract. 3-339
Figure 3-13. HSUBPS: Packed Single-FP Horizontal Subtract. 3-343
Figure 3-14. MOVDDUP: Move One Double-FP and Duplicate 3-473
Figure 3-15. MOVSHDUP: Move Packed Single-FP High and Duplicate. 3-503
Figure 3-16. MOVSLDUP: Move Packed Single-FP Low and Duplicate 3-506
Figure 4-1. Operation of the PACKSSDW Instruction Using 64-bit Operands. 4-17
Figure 4-2. PMADDWD Execution Model Using 64-bit Operands 4-55
Figure 4-3. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands . . . 4-72
Figure 4-4. PMULLU Instruction Operation Using 64-bit Operands 4-78
Figure 4-5. PSADBW Instruction Operation Using 64-bit Operands. 4-98
Figure 4-6. PSHUFD Instruction Operation. 4-100
Figure 4-7. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand 4-109
Figure 4-8. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand 4-113
Figure 4-9. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand 4-118
Figure 4-10. PUNPCKHBW Instruction Operation Using 64-bit Operands. 4-133
Figure 4-11. PUNPCKLBW Instruction Operation Using 64-bit Operands 4-137
Figure 4-12. SHUFPD Shuffle Operation 4-199
Figure 4-13. SHUFPS Shuffle Operation 4-201
Figure 4-14. UNPCKHPD Instruction High Unpack and Interleave Operation 4-255
Figure 4-15. UNPCKHPS Instruction High Unpack and Interleave Operation 4-257
Figure 4-16. UNPCKLPD Instruction Low Unpack and Interleave Operation 4-259
Figure 4-17. UNPCKLPS Instruction Low Unpack and Interleave Operation 4-261
Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3). A-13
Figure B-1. General Machine Instruction Format B-1

PAGE

TABLES

PAGE

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte 2-6
Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte 2-7
Table 2-3. 32-Bit Addressing Forms with the SIB Byte 2-8
Table 3-1. Register Encodings Associated with the +rb, +rw, and +rd Nomenclature. . . . 3-2
Table 3-2. IA-32 General Exceptions. 3-13
Table 3-3. x87 FPU Floating-Point Exceptions 3-14
Table 3-4. SIMD Floating-Point Exceptions. 3-15
Table 3-5. Decision Table for CLI Results 3-82
Table 3-6. Comparison Predicate for CMPPD and CMPPS Instructions. 3-91
Table 3-7. Pseudo-Op and CMPPD Implementation 3-92
Table 3-8. Pseudo-Ops and CMPPS. 3-96
Table 3-9. Pseudo-Ops and CMPSD. 3-103
Table 3-10. Pseudo-Ops and CMPSS. 3-107
Table 3-11. Information Returned by CPUID Instruction 3-121
Table 3-12. Highest CPUID Source Operand for IA-32 Processors 3-123
Table 3-13. Processor Type Field 3-125
Table 3-14. More on Extended Feature Information Returned in the ECX Register 3-127
Table 3-15. More on Feature Information Returned in the EDX Register 3-128
Table 3-16. Encoding of Cache and TLB Descriptors 3-132
Table 3-17. Processor Brand String Returned with Pentium 4 Processor. 3-135
Table 3-18. Mapping of Brand Indices and IA-32 Processor Brand Strings 3-138
Table 3-19. DIV Action. 3-194
Table 3-20. Results Obtained from F2XM1 3-209
Table 3-21. Results Obtained from FABS 3-211
Table 3-22. FADD/FADDP/FIADD Results 3-213
Table 3-23. FBSTP Results. 3-217
Table 3-24. FCHS Results. 3-220
Table 3-25. FCOM/FCOMP/FCOMPP Results 3-225
Table 3-26. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results 3-228
Table 3-27. FCOS Results. 3-231
Table 3-28. FDIV/FDIVP/FIDIV Results. 3-235
Table 3-29. FDIVR/FDIVRP/FIDIVR Results. 3-238
Table 3-30. FICOM/FICOMP Results 3-241
Table 3-31. FIST/FISTP Results 3-248
Table 3-32. FISTTP Results 3-251
Table 3-33. FMUL/FMULP/FIMUL Results 3-262
Table 3-34. FPATAN Results. 3-265
Table 3-35. FPREM Results 3-267
Table 3-36. FPREM1 Results 3-270
Table 3-37. FPTAN Results. 3-273
Table 3-38. FSCALE Results. 3-281
Table 3-39. FSIN Results. 3-283
Table 3-40. FSINCOS Results. 3-285
Table 3-41. FSQRT Results. 3-287
Table 3-42. FSUB/FSUBP/FISUB Results. 3-300
Table 3-43. FSUBR/FSUBRP/FISUBR Results. 3-303
Table 3-44. FTST Results 3-305
Table 3-45. FUCOM/FUCOMP/FUCOMPP Results 3-307
Table 3-46. FXAM Results. 3-311

PAGE

Table 3-47. Layout of FXSAVE and FXRSTOR Memory Region 3-319
Table 3-48. Recreating FSAVE Format 3-323
Table 3-49. FYL2X Results 3-327
Table 3-50. FYL2XP1 Results 3-329
Table 3-51. IDIV Results 3-346
Table 3-52. Decision Table 3-360
Table 3-53. Segment and Gate Types 3-393
Table 3-54. Address and Operand Size Attributes 3-402
Table 3-55. Segment and Gate Descriptor Types 3-424
Table 3-56. MUL Results 3-525
Table 4-1. Repeat Prefixes 4-165
Table 4-2. Decision Table for STI Results 4-221
Table 4-3. MSRs Used By the SYSENTER and SYSEXIT Instructions 4-239
Table A-1. Notes on Instruction Encoding in Opcode Map Tables. A-6
Table A-2. One-byte Opcode Map† †† A-7
Table A-3. Two-byte Opcode Map (First Byte is 0FH) A-9
Table A-4. Opcode Extensions for One- and Two-byte Opcodes by Group Number A-13
Table A-5. D8 Opcode Map When ModR/M Byte is Within 00H to BFH1 ..... A-15
Table A-6. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH1 ..... A-16
Table A-7. D9 Opcode Map When ModR/M Byte is Within 00H to BFH1. ..... A-17
Table A-8. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH1 ..... A-17
Table A-9. DA Opcode Map When ModR/M Byte is Within 00H to BFH1 ..... A-18
Table A-10. DA Opcode Map When ModR/M Byte is Outside 00H to BFH1 ..... A-18
Table A-11. DB Opcode Map When ModR/M Byte is Within 00H to BFH1 ..... A-19
Table A-12. DB Opcode Map When ModR/M Byte is Outside 00H to BFH1 ..... A-19
Table A-13. DC Opcode Map When ModR/M Byte is Within 00H to BFH1 ..... A-20
Table A-14. DC Opcode Map When ModR/M Byte is Outside 00H to BFH4 ..... A-20
Table A-15. DD Opcode Map When ModR/M Byte is Within 00H to BFH1 ..... A-21
Table A-16. DD Opcode Map When ModR/M Byte is Outside 00H to BFH1 ..... A-21
Table A-17. DE Opcode Map When ModR/M Byte is Within 00H to BFH1 ..... A-22
Table A-18. DE Opcode Map When ModR/M Byte is Outside 00H to BFH1 ..... A-22
Table A-19. DF Opcode Map When ModR/M Byte is Within 00H to BFH1 ..... A-23
Table A-20. 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 (tttn) Field. B-5
Table B-9. Encoding of Operation Direction (d) Bit B-6
Table B-10. Notes on Instruction Encoding B-6
Table B-11. General Purpose Instruction Formats and Encodings B-6
Table B-12. Pentium Family Instruction Formats and Encodings B-19
Table B-13. Encoding of Granularity of Data Field (gg) B-20
Table B-14. MMX Instruction Formats and Encodings. B-20
Table B-15. Formats and Encodings of P6 Family Instructions B-24
Table B-16. Formats and Encodings of SSE Floating-Point Instructions B-25
Table B-17. Formats and Encodings of SSE Integer Instructions B-31

PAGE

Table B-18. Format and Encoding of SSE Cacheability and Memory Ordering Instructions B-32
Table B-19. Encoding of Granularity of Data Field (gg) B-33
Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions B-33
Table B-21. Formats and Encodings of SSE2 Integer Instructions B-40
Table B-22. Format and Encoding of SSE2 Cacheability Instructions B-45
Table B-23. Formats and Encodings of SSE3 Floating-Point Instructions B-46
Table B-24. Formats and Encodings for SSE3 Event Management Instructions B-47
Table B-25. Formats and Encodings for SSE3 Integer and Move Instructions B-47
Table B-26. General Floating-Point Instruction Formats B-48
Table B-27. Floating-Point Instruction Formats and Encodings B-49
Table C-1. Simple Intrinsics C-3
Table C-2. Composite Intrinsics C-32

1

About This Manual

intel®

CHAPTER 1
ABOUT THIS MANUAL

The *IA-32 Intel® Architecture Software Developer's Manual, Volumes 2A & 2B*: *Instruction Set Reference* (Order Numbers 253666 and 253667) are part of a set that describes the architecture and programming environment of all IA-32 Intel Architecture processors. Other volumes in this set are:

- The *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture* (Order Number 253665).
- The *IA-32 Intel Architecture Software Developer's Manual, Volume 3*: *System Programing Guide* (Order Number 253668).

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, Volumes 2A & 2B* describe the instructions set of the processor and the opcode structure. These 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 and IA-32 processor compatibility information. This volume is aimed at operating-system and BIOS designers.

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® processors, the P6 family processors, the Pentium 4 processors, the Intel® Xeon™ processors, and the Pentium M processors. The P6 family processors are those IA-32 processors based on the P6 family microarchitecture, which include the Pentium Pro, Pentium II, and Pentium III processors. The Pentium 4 and Intel Xeon processors are based on the Intel NetBurst® microarchitecture.

1.2. OVERVIEW OF THE *IA-32 INTEL® ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUMES 2A & 2B: INSTRUCTION SET REFERENCE*

A description of *IA-32 Intel Architecture Software Developer's Manual, Volumes 2A & 2B* content 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, A-M. Describes IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. General-purpose, x87 FPU, Intel MMX™ technology, SSE/SSE2/SSE3 extensions, and system instructions are included.

Chapter 4 — Instruction Set Reference, N-Z. This chapter continues the description of IA-32 instructions started in Chapter 3. It provides the balance of the alphabetized list of instructions and starts *IA-32 Intel Architecture Software Developer's Manual, Volume 2B*.

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

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

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

1.3. NOTATIONAL CONVENTIONS

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

1.3.1. Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 proces-

sors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

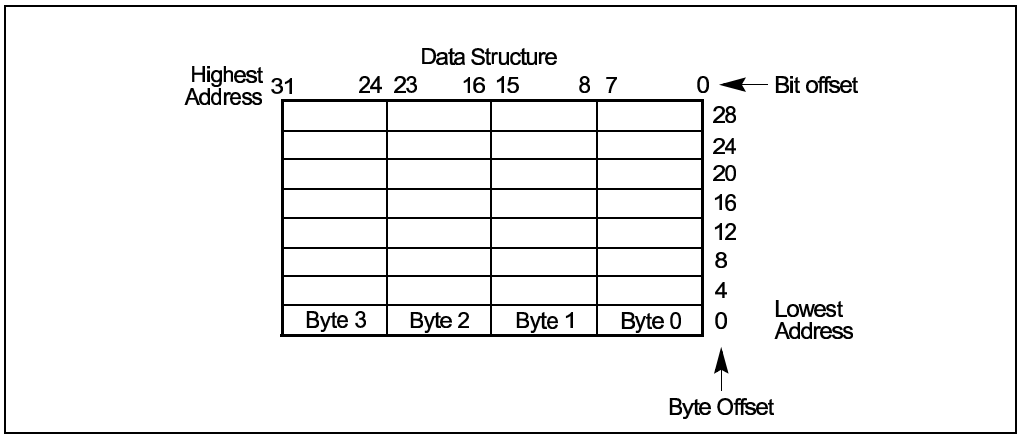

Figure 1-1. Bit and Byte Order

1.3.2. Reserved Bits and Software Compatibility

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

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

NOTE

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

1.3.3. Instruction Operands

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

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands *argument1*, *argument2*, and *argument3* are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

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

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

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

1.3.4. Hexadecimal and Binary Numbers

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

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

1.3.5. Segmented Addressing

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

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

Segment-register:Byte-address

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

```
DS:FF79H
```

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

```
CS:EIP
```

1.3.6. Exceptions

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

```
#PF(fault code)
```

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

```
#GP(0)
```

1.4. RELATED LITERATURE

Literature related to IA-32 processors is listed on-line at this link:

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

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

See also:

- The data sheet for a particular Intel IA-32 processor
- The specification update for a particular Intel IA-32 processor
- AP-485, *Intel Processor Identification and the CPUID Instruction*, Order Number 241618
- *IA-32 Intel® Architecture Optimization Reference Manual*, Order Number 248966

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 format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes of up to three 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).

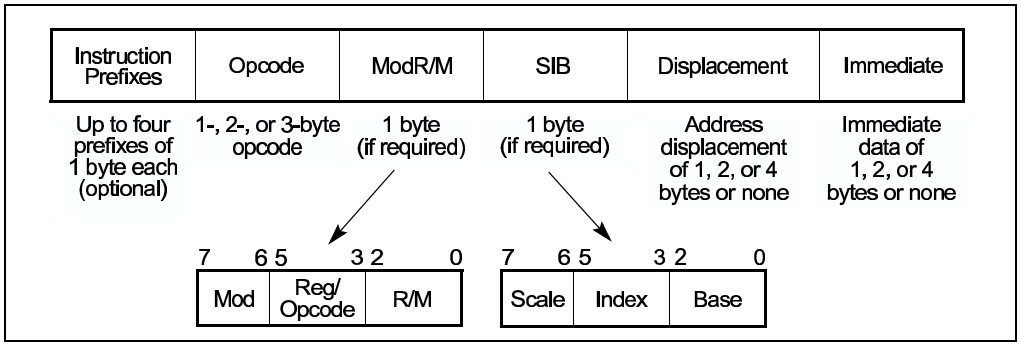

Figure 2-1. IA-32 Instruction Format

2.2. SUMMARY OF INSTRUCTION PREFIXES

The instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, one prefix may be used from each of four groups (Groups 1, 2, 3, 4) and be placed in any order.

NOTE

Developers must not rely on the absence or characteristics of any features or instructions marked reserved or undefined. Improper use of reserved features may cause code to exhibit unpredictable behavior or fail when the code is running on an Intel processor.

- Group 1
  - Lock and repeat prefixes:
    - F0H—LOCK
    - F2H—REPNE/REPNZ (used only with string instructions; when used with the escape opcode 0FH, this prefix is treated as a mandatory prefix for some SIMD instructions)
    - F3H—REP or REPE/REPZ (used only with string instructions; when used with the escape opcode 0FH, this prefix is treated as an mandatory prefix for some SIMD 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 J*cc* instructions)
    - 3EH—Branch taken (used only with J*cc* instructions)
- Group 3
    - 66H—Operand-size override prefix (when used with the escape opcode 0FH, this is treated as a mandatory prefix for some SIMD instructions)

- Group 4
    - 67H—Address-size override prefix

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

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Their use, followed by 0FH, is treated as a mandatory prefix by a number of SSE/SSE2/SSE3 instructions. Use of repeat prefixes and/or undefined opcodes with other IA-32 instructions is reserved; such use may cause unpredictable behavior.

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

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size. Use of 66H followed by 0FH is treated as a mandatory prefix by some SSE/SSE2/SSE3 instructions. Other use of the 66H prefix with MMX/SSE/SSE2/SSE3 instructions is reserved; such use may cause unpredictable behavior.

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

2.3. OPCODES

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

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

- An escape opcode byte 0FH as the primary opcode and a second opcode byte
- A mandatory prefix (66FH, F2H, F3H), an escape opcode byte, and a second opcode byte

For example, CVTDQ2PD consists of the following byte sequence: F3 OF E6. The first byte of this expression is a mandatory prefix for SSE/SSE2/SSE3 instructions. It is not considered as a repeat prefix. Note that all three byte opcodes are reserved.

The ModR/M byte consists of three bit fields (see Section 2.4.). In addition to the *reg* field being treated as an extended opcode field for some instructions, some patterns of the other two bit

fields in the ModR/M byte may also be used to express opcode information. Using undefined expression of the primary opcode bytes, and/or undefined expression in the opcode extension field in the ModR/M byte, and/or undefined expression in other bit fields of the ModR/M byte is reserved. Valid opcode expressions are defined in Appendix A and Appendix B. Use of any of reserved opcode expression can cause unpredictable behavior.

2.4. MODR/M AND SIB BYTES

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

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

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

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

See Section 2.6., "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 the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can require 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 Table 2-1 through Table 2-3. The 16-bit addressing forms specified by the ModR/M byte are in Table 2-1, 32-bit addressing forms are in Table 2-2. Table 2-3 shows the 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid instruction encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the first column (labeled "Effective Address") lists 32 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 provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology, and XMM registers. For example, the first register-encoding (Mod = 11B, R/M = 000B) indicates general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute. These may select either the EAX register (32 bits) or AX register (16 bits).

The second and third columns in Table 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 Table 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 technology, 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 technology 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 Table 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. The row specifies bits 0, 1 and 2; 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[1] EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16[2]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		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/MM1/XMM1		001	C1	C9	D1	D9	EQ	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/AHMM4/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 default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

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

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

NOTES:

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

Table 2-3 is organized 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 eight 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	89	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 the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

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

3

Instruction Set Reference, A-M

intel®

CHAPTER 3
INSTRUCTION SET REFERENCE, A-M

This chapter describes the IA-32 instruction set, including the general-purpose, x87 FPU, MMX, SSE/SSE2/SSE3, and system instructions. The descriptions in this chapter are arranged in alphabetical order (A-M). This discussion is continued in Chapter 4 for the balance of the IA-32 instruction set (N-Z). See also Chapter 4, *IA-32 Intel Architecture Software Developer's Manual, Volume 2B*.

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. Each heading is followed by a usage description.

CMC—Complement Carry Flag [this is an example with usage description]

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 3-1.
- **+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/SSE2/SSE3 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^{th} element from the top of the FPU register stack. ($i \leftarrow 0$ through 7)
- **mm**—An MMX technology register. The 64-bit MMX technology registers are: MM0 through MM7.
- **mm/m32**—The low order 32 bits of an MMX technology register or a 32-bit memory operand. The 64-bit MMX technology registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **mm/m64**—An MMX technology register or a 64-bit memory operand. The 64-bit MMX technology 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 ← B; indicates that the value of B is assigned to A.
- The symbols =, ≠, ≥, and ≤ are relational operators used to compare two values, meaning equal, not equal, greater or equal, and less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.

- The expression "<< COUNT" and ">> COUNT" indicates that the destination operand should be shifted left or right, 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 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)**—Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value –10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte**—Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateSignedDwordToSignedWord**—Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than –32768, it is represented by the saturated

value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

- **SaturateSignedWordToUnsignedByte**—Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte**—Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord**—Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte**—Represents the result of an operation as a signed 8-bit value. If the result is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord**—Represents the result of an operation as a signed 16-bit value. If the result is less than zero, it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).
- **RoundTowardsZero()**—Returns the operand rounded towards zero to the nearest integral value.
- **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 Chapter 4 for more information on the push operation.
- **Pop()** removes the value from the top of the stack and returns it. The statement EAX ← Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute. See the "Operation" section in Chapter 4, "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 Bit [EAX, 21] is illustrated in Figure 3-1.

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.

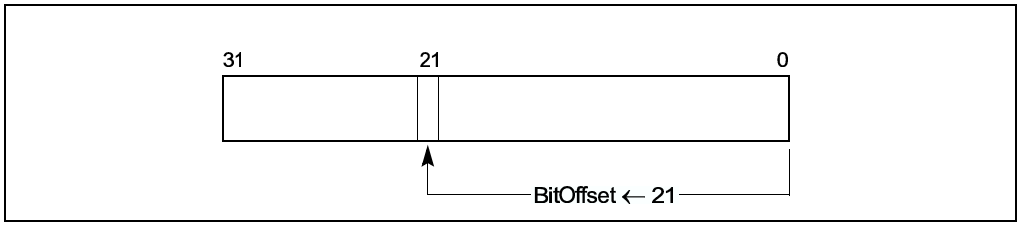

Figure 3-1. Bit Offset for BIT[EAX,21]

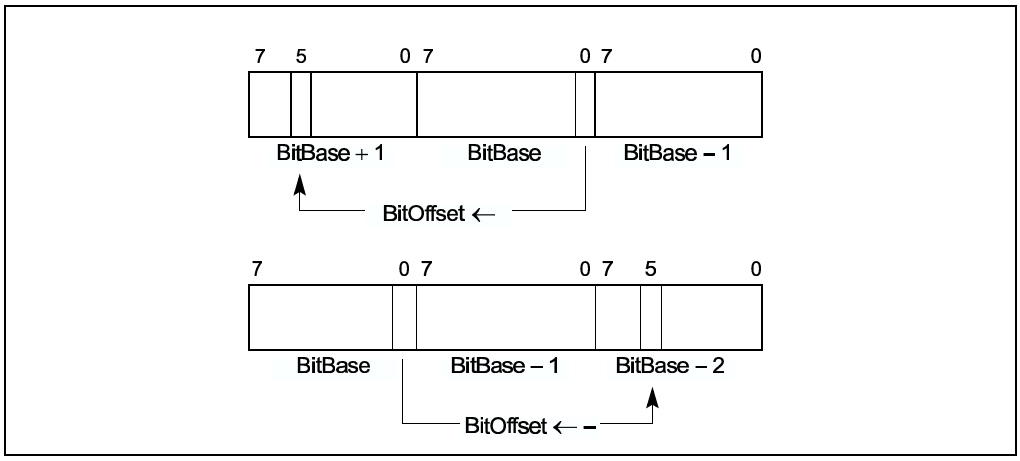

Figure 3-2. Memory Bit Indexing

3.1.3. Intel® C/C++ Compiler Intrinsics 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 executable run faster.

The following sections discuss the intrinsics API and the MMX technology and SIMD floating-point intrinsics. Each intrinsic equivalent is listed with the instruction description. There may be additional intrinsics that do not have an instruction equivalent. It is strongly recommended that the reader reference the compiler documentation for the complete list of supported intrinsics.

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

3.1.3.2. MMX™ TECHNOLOGY INTRINSICS

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

- Use __m64 data only on the left-hand side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions ("+", ">>", and so on).
- Use __m64 objects in aggregates, such as unions to access the byte elements and structures; the address of an __m64 object may be taken.
- Use __m64 data only with the MMX technology intrinsics described in this manual and 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/SSE2/SSE3 INTRINSICS

SSE/SSE2/SSE3 intrinsics all make use of the XMM registers of the Pentium III, Pentium 4, and Intel Xeon processors. There are three data types supported by these intrinsics: __m128, __m128d, and __m128i.

- The __m128 data type is used to represent the contents of an XMM register used by an SSE intrinsic. This is either four packed single-precision floating-point values or a scalar single-precision floating-point value.
- The __m128d data type holds two packed double-precision floating-point values or a scalar double-precision floating-point value.
- The __m128i data type can hold sixteen byte, eight word, or four doubleword, or two quadword integer values.

The compiler aligns __m128, __m128d, and __m128i local and global data to 16-byte boundaries on the stack. To align integer, float, or double arrays, 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, SSE3, and MMX technology when writing programs with the intrinsics.

Keep the following 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 may not match the expected behavior of the corresponding assembly instructions.

For a more detailed description of each intrinsic and additional information related to its usage, refer to 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.

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/SSE2/SSE3 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 #IA	Floating-point invalid operation: - Stack overflow or underflow - Invalid arithmetic operation	 - x87 FPU stack overflow or underflow - Invalid FPU arithmetic operation
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result (precision)	Inexact result (precision)

3.1.10. SIMD Floating-Point Exceptions

The "SIMD Floating-Point Exceptions" section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in an 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

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

3.2. INSTRUCTION REFERENCE

The remainder of this chapter provides detailed descriptions of IA-32 instruction.

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

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 set 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 *ib*	(No mnemonic)	Adjust AX before division to number base *imm8*

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 *ib*	(No mnemonic)	Adjust AX after multiply to number base *imm8*

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 AAM 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 decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top 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 set to 0. The OF, SF, ZF, and PF flags are undefined.

Exceptions (All Operating Modes)

None.

ADC—Add with Carry

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

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.

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

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.

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 *xmm1, xmm2/m128*	Add packed double-precision floating-point values from *xmm2/m128* to *xmm1*.

Description

Performs an 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 an 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 58 /r	ADDPS *xmm1, xmm2/m128*	Add packed single-precision floating-point values from *xmm2/m128* to *xmm1*.

Description

Performs an 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 an 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *xmm1*, *xmm2/m64*	Add the low double-precision floating-point value from *xmm2/m64* to *xmm1*.

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.

#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 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 *xmm1, xmm2/m32*	Add the low single-precision floating-point value from *xmm2/m32* to *xmm1*.

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.

#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 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.

ADDSUBPD: Packed Double-FP Add/Subtract

Opcode	Instruction	Description
66,0F,D0,/r	ADDSUBPD *xmm1*, *xmm2/m128*	Add/Subtract packed DP FP numbers from *xmm2/m128* to *xmm1*.

Description

Adds the double-precision floating-point values in the high quadword of the source and destination operands and stores the result in the high quadword of the destination operand.

Subtracts the double-precision floating-point value in the low quadword of the source operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand. See

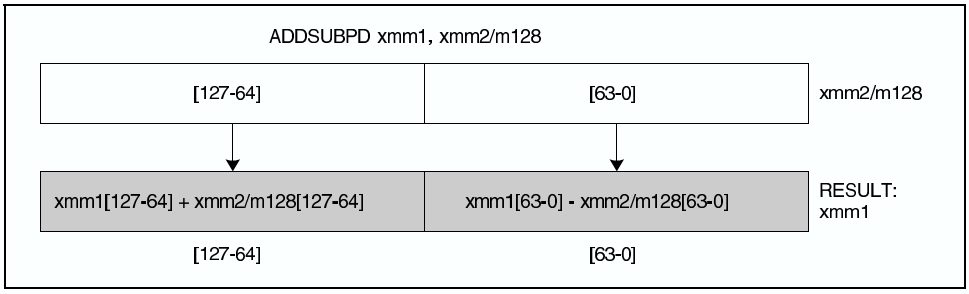

Figure 3-3. ADDSUBPD: Packed Double-FP Add/Subtract

Operation

```
xmm1[63-0]    = xmm1[63-0]    - xmm2/m128[63-0];
xmm1[127-64] = xmm1[127-64] + xmm2/m128[127-64];
```

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPD __m128d _mm_addsub_pd(__m128d a, __m128d b)

Exceptions

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

Numeric 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 a 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 bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0);
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Virtual 8086 Mode Exceptions

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

#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.
#PF(fault-code)	For a page fault.

ADDSUBPS: Packed Single-FP Add/Subtract

Opcode	Instruction	Description
F2,0F,D0,/r	ADDSUBPS *xmm1*, *xmm2/m128*	Add/Subtract packed SP FP numbers from *xmm2/m128* to *xmm1*.

Description

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

Subtracts the even-numbered single-precision floating-point values in the source operand from the corresponding single-precision floating values in the destination operand; stores the result into the even-numbered values of the destination operand. See Figure 3-4.

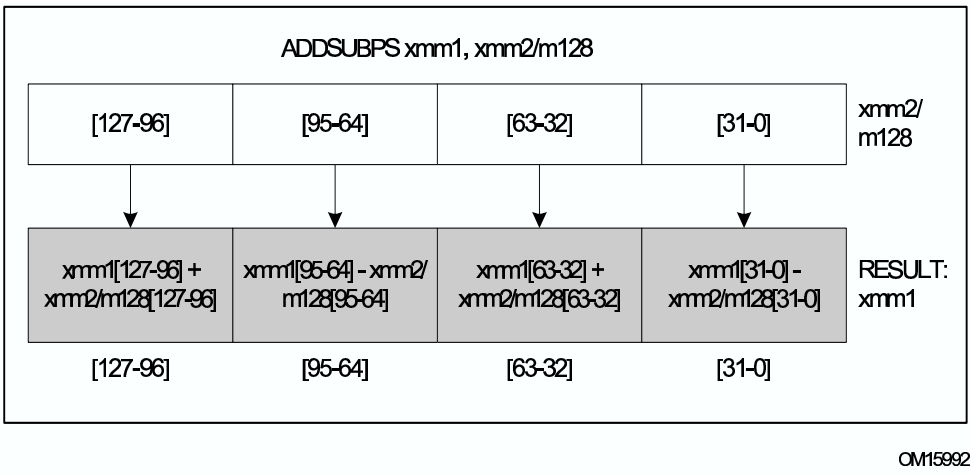

Figure 3-4. ADDSUBPS: Packed Single-FP Add/Subtract

Operation

```
xmm1[31-0]   = xmm1[31-0]   - xmm2/m128[31-0];
xmm1[63-32]  = xmm1[63-32]  + xmm2/m128[63-32];
xmm1[95-64]  = xmm1[95-64]  - xmm2/m128[95-64];
xmm1[127-96] = xmm1[127-96] + xmm2/m128[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPS __m128 _mm_addsub_ps(__m128 a, __m128 b)

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).

If CR4.OSFXSR(bit 9) = 0.

If CPUID.SSE3(ECX bit 0) = 0.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.
#PF(fault-code)	For a page fault.

AND—Logical AND

Opcode	Instruction	Description
24 *ib*	AND AL,*imm8*	AL AND *imm8*
25 *iw*	AND AX,*imm16*	AX AND i*mm16*
25 *id*	AND EAX,*imm32*	EAX AND *imm32*
80 /4 *ib*	AND *r/m8,imm8*	*r/m8* AND *imm8*
81 /4 *iw*	AND *r/m16,imm16*	*r/m16* AND *imm16*
81 /4 *id*	AND *r/m32,imm32*	*r/m32* AND *imm32*
83 /4 *ib*	AND *r/m16,imm8*	*r/m16* AND *imm8 (sign-extended)*
83 /4 *ib*	AND *r/m32,imm8*	*r/m32* AND *imm8 (sign-extended)*
20 /*r*	AND *r/m8,r8*	*r/m8* AND *r8*
21 /*r*	AND *r/m16,r16*	*r/m16* AND *r16*
21 /*r*	AND *r/m32,r32*	*r/m32* AND *r32*
22 /*r*	AND *r8,r/m8*	*r8* AND *r/m8*
23 /*r*	AND *r16,r/m16*	*r16* AND *r/m16*
23 /*r*	AND *r32,r/m32*	*r32* AND *r/m32*

Description

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

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

Operation

```
DEST ← DEST AND SRC;
```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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 *xmm1*, *xmm2/m128*	Bitwise logical AND of *xmm2/m128* and *xmm1*.

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 a 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 SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

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

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

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

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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.

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

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

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

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 a 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 SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

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

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

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

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 a 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.
A	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

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

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, it is set to 0.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.

	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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 *r16, m16&16*	Check if *r16* (array index) is within bounds specified by *m16&16*
62 /r	BOUND *r32, m32&32*	Check if *r32* (array index) is within bounds specified by *m32&32*

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 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 > UpperBound)
    (* 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.

#SS(0)	If a memory operand effective address is outside the SS 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 *r16,r/m16*	Bit scan forward on *r/m16*
0F BC	BSF *r32,r/m32*	Bit scan forward on *r/m32*

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is 0, the content of the destination operand is undefined.

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.

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
0F BD	BSR *r16,r/m16*	Bit scan reverse on *r/m16*
0F BD	BSR *r32,r/m32*	Bit scan reverse on *r/m32*

Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined.

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.

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 *r/m16,r16*	Store selected bit in CF flag
0F A3	BT *r/m32,r32*	Store selected bit in CF flag
0F BA /4 *ib*	BT *r/m16,imm8*	Store selected bit in CF flag
0F BA /4 *ib*	BT *r/m32,imm8*	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:

Effective Address + (4 * (BitOffset DIV 32))

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

Effective Address + (2 * (BitOffset DIV 16))

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

Operation

CF ← Bit(BitBase, BitOffset)

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 *r/m16,r16*	Store selected bit in CF flag and complement
0F BB	BTC *r/m32,r32*	Store selected bit in CF flag and complement
0F BA /7 *ib*	BTC *r/m16,imm8*	Store selected bit in CF flag and complement
0F BA /7 *ib*	BTC *r/m32,imm8*	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 ← Bit(BitBase, BitOffset)
Bit(BitBase, BitOffset) ← NOT Bit(BitBase, BitOffset);
```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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 *r/m16,r16*	Store selected bit in CF flag and clear
0F B3	BTR *r/m32,r32*	Store selected bit in CF flag and clear
0F BA /6 *ib*	BTR *r/m16,imm8*	Store selected bit in CF flag and clear
0F BA /6 *ib*	BTR *r/m32,imm8*	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 non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#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 *r/m16,r16*	Store selected bit in CF flag and set
0F AB	BTS *r/m32,r32*	Store selected bit in CF flag and set
0F BA /5 *ib*	BTS *r/m16,imm8*	Store selected bit in CF flag and set
0F BA /5 *ib*	BTS *r/m32,imm8*	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 non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

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.

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, 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.

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.

Note that a call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. 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 procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction, is 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 the 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 tasks 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.

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 17, *Mixing 17-Bit and 32-Bit Code*, in *IA-32 Intel Architecture Software Developer's Manual, Volume 3* for more information.

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;
                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);
        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;
    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 *)
        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 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;
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.

Protected Mode Exceptions

#GP(0)	If the target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is null.
	If the code segment selector in the gate is null.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If a code segment or gate or TSS selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.

	If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.
	If the segment selector from a call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a call gate is greater than the CPL.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.
	If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.
	If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.
#NP(selector)	If a code segment, data segment, 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 ← 0;

Flags Affected

The CF flag is set 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 ← 0;

Flags Affected

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

Exceptions (All Operating Modes)

None.

CLFLUSH—Flush Cache Line

Opcode	Instruction	Description
0F AE /7	CLFLUSH *m8*	Flushes cache line containing *m8*.

Description

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

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see 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, and WT memory types). PREFETCH*h* instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH*h* instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to 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)
```

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

GP(0)	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

If protected-mode virtual interrupts are not enabled, CLI clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no affect on the generation of exceptions and NMI interrupts.

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

Table 3-5 indicates the action of the CLI instruction depending on the processor operating mode and the CPL/IOPL of the running program or procedure.

Table 3-5. Decision Table for CLI Results

PE	VM	IOPL	CPL	PVI	VIP	VME	CLI Result
0	X	X	X	X	X	X	**IF = 0**
1	0	≥ CPL	X	X	X	X	**IF = 0**
1	0	< CPL	3	1	X	X	**VIF = 0**
1	0	< CPL	< 3	X	X	X	**GP Fault**
1	0	< CPL	X	0	X	X	**GP Fault**
1	1	3	X	X	X	X	**IF = 0**
1	1	< 3	X	X	X	1	**VIF = 0**
1	1	< 3	X	X	X	0	**GP Fault**
X = This setting has no impact.							

Operation

```
IF PE = 0
    THEN
        IF ← 0; (* Reset Interrupt Flag *)
    ELSE
        IF VM = 0;
            THEN
                IF IOPL ≥ CPL
                    THEN
                        IF ← 0; (* Reset Interrupt Flag *)
                ELSE
                    IF ((IOPL < CPL) AND (CPL < 3) AND (PVI = 1))
                        THEN
```

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

Flags Affected

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

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

Protected Mode Exceptions

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

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 current privilege level is not 0.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)	CLTS is not recognized in virtual-8086 mode.

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.

CMOV*cc*—Conditional Move

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

Opcode	Instruction	Description
0F 41 /r	CMOVNO *r16, r/m16*	Move if not overflow (OF=0).
0F 41 /r	CMOVNO *r32, r/m32*	Move if not overflow (OF=0).
0F 4B /r	CMOVNP *r16, r/m16*	Move if not parity (PF=0).
0F 4B /r	CMOVNP *r32, r/m32*	Move if not parity (PF=0).
0F 49 /r	CMOVNS *r16, r/m16*	Move if not sign (SF=0).
0F 49 /r	CMOVNS *r32, r/m32*	Move if not sign (SF=0).
0F q5 /r	CMOVNZ *r16, r/m16*	Move if not zero (ZF=0).
0F 45 /r	CMOVNZ *r32, r/m32*	Move if not zero (ZF=0).
0F 40 /r	CMOVO *r16, r/m16*	Move if overflow (OF=1).
0F 40 /r	CMOVO *r32, r/m32*	Move if overflow (OF=1).
0F 4A /r	CMOVP *r16, r/m16*	Move if parity (PF=1).
0F 4A /r	CMOVP *r32, r/m32*	Move if parity (PF=1).
0F 4A /r	CMOVPE *r16, r/m16*	Move if parity even (PF=1).
0F 4A /r	CMOVPE *r32, r/m32*	Move if parity even (PF=1).
0F 4B /r	CMOVPO *r16, r/m16*	Move if parity odd (PF=0).
0F 4B /r	CMOVPO *r32, r/m32*	Move if parity odd (PF=0).
0F 48 /r	CMOVS *r16, r/m16*	Move if sign (SF=1).
0F 48 /r	CMOVS *r32, r/m32*	Move if sign (SF=1).
0F 44 /r	CMOVZ *r16, r/m16*	Move if zero (ZF=1).
0F 44 /r	CMOVZ *r32, r/m32*	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.

The CMOV*cc* instructions were introduced in the P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOV*cc* instructions are supported by checking the processor's feature information with the CPUID

instruction (see "COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS" in this chapter).

Operation

```
temp ← SRC
IF condition TRUE
    THEN
        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.
#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 *ib*	CMP AL, *imm8*	Compare *imm8* with AL.
3D *iw*	CMP AX, *imm16*	Compare *imm16* with AX.
3D *id*	CMP EAX, *imm32*	Compare *imm32* with EAX.
80 */7 ib*	CMP *r/m8, imm8*	Compare *imm8* with *r/m8.*
81 */7 iw*	CMP *r/m16, imm16*	Compare *imm16* with *r/m16.*
81 */7 id*	CMP *r/m32,imm32*	Compare *imm32* with *r/m32.*
83 */7 ib*	CMP *r/m16,imm8*	Compare *imm8* with *r/m16.*
83 */7 ib*	CMP *r/m32,imm8*	Compare *imm8* with *r/m32.*
38 */r*	CMP *r/m8,r8*	Compare *r8* with *r/m8.*
39 */r*	CMP *r/m16,r16*	Compare *r16* with *r/m16.*
39 */r*	CMP *r/m32,r32*	Compare *r32* with *r/m32.*
3A */r*	CMP *r8,r/m8*	Compare *r/m8* with *r8.*
3B */r*	CMP *r16,r/m16*	Compare *r/m16* with *r16.*
3B */r*	CMP *r32,r/m32*	Compare *r/m32* with *r32.*

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 (J*cc*), condition move (CMOV*cc*), or SET*cc* instruction. The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. Appendix B, *EFLAGS Condition Codes*, in the *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.
	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 *xmm1*, *xmm2/m128*, *imm8*	Compare packed double-precision floating-point values in *xmm2/m128* and *xmm1* using imm8 as comparison predicate.

Description

Performs an 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 3-6); bits 4 through 7 of the immediate are reserved.

Table 3-6. Comparison Predicate for CMPPD and CMPPS Instructions

Predicate	imm8 Encoding	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Operand Signals Invalid
EQ	000B	Equal	A = B		False	No
LT	001B	Less-than	A < B		False	Yes
LE	010B	Less-than-or-equal	A ≤ B		False	Yes
		Greater than	A > B	Swap Operands, Use LT	False	Yes
		Greater-than-or-equal	A ≥ B	Swap Operands, Use LE	False	Yes
UNORD	011B	Unordered	A, B = Unordered		True	No
NEQ	100B	Not-equal	A ≠ B		True	No
NLT	101B	Not-less-than	NOT(A < B)		True	Yes
NLE	110B	Not-less-than-or-equal	NOT(A ≤ B)		True	Yes
		Not-greater-than	NOT(A > B)	Swap Operands, Use NLT	True	Yes

Table 3-6. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

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

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that the 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 3-6 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction. See Table 3-7.

Table 3-7. Pseudo-Op and CMPPD Implementation

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

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

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

SIMD Floating-Point Exceptions

Invalid if SNaN operand and 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *xmm1, xmm2/m128, imm8*	Compare packed single-precision floating-point values in *xmm2/mem* and *xmm1* using *imm8* as comparison predicate.

Description

Performs an 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 3-6); 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; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Some of the comparisons listed in Table 3-6 (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 3-6 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction. See Table 3-8.

Table 3-8. Pseudo-Ops and CMPPS

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

The greater-than relations 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 DEST95-64] ← 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	__m128 _mm_cmpeq_ps(__m128 a, __m128 b)
CMPPS for less-than	__m128 _mm_cmplt_ps(__m128 a, __m128 b)
CMPPS for less-than-or-equal	__m128 _mm_cmple_ps(__m128 a, __m128 b)
CMPPS for greater-than	__m128 _mm_cmpgt_ps(__m128 a, __m128 b)
CMPPS for greater-than-or-equal	__m128 _mm_cmpge_ps(__m128 a, __m128 b)
CMPPS for inequality	__m128 _mm_cmpneq_ps(__m128 a, __m128 b)
CMPPS for not-less-than	__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)
CMPPS for not-greater-than	__m128 _mm_cmpngt_ps(__m128 a, __m128 b)
CMPPS for not-greater-than-or-equal	__m128 _mm_cmpnge_ps(__m128 a, __m128 b)
CMPPS for ordered	__m128 _mm_cmpord_ps(__m128 a, __m128 b)
CMPPS for unordered	__m128 _mm_cmpunord_ps(__m128 a, __m128 b)
CMPPS for not-less-than-or-equal	__m128 _mm_cmpnle_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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.

CMPS/CMPSB/CMPSW/CMPSD—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	CMPSD	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 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 CMPSD (doubleword comparison).

After the comparison, the (E)SI and (E)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register increment; if the DF flag is 1, the (E)SI and (E)DI registers decrement.) The registers increment or decrement 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 Chapter 4 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;
```

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 *xmm1*, *xmm2/m64*, *imm8*	Compare low double-precision floating-point value in *xmm2/m64* and *xmm1* using *imm8* 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 3-6); 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; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Some of the comparisons listed in Table 3-6 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 3-6 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction. See Table 3-9.

Table 3-9. Pseudo-Ops and CMPSD

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

The greater-than relations 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] ← 0000000000000000H; FI;
* DEST[127-64] remains unchanged *;
```

Intel C/C++ Compiler Intrinsic Equivalents

CMPSD for equality	__m128d _mm_cmpeq_sd(__m128d a, __m128d b)
CMPSD for less-than	__m128d _mm_cmplt_sd(__m128d a, __m128d b)
CMPSD for less-than-or-equal	__m128d _mm_cmple_sd(__m128d a, __m128d b)
CMPSD for greater-than	__m128d _mm_cmpgt_sd(__m128d a, __m128d b)
CMPSD for greater-than-or-equal	__m128d _mm_cmpge_sd(__m128d a, __m128d b)
CMPSD for inequality	__m128d _mm_cmpneq_sd(__m128d a, __m128d b)

CMPSD for not-less-than	__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)
CMPSD for not-greater-than	__m128d _mm_cmpngt_sd(__m128d a, __m128d b)
CMPSD for not-greater-than-or-equal	__m128d _mm_cmpnge_sd(__m128d a, __m128d b)
CMPSD for ordered	__m128d _mm_cmpord_sd(__m128d a, __m128d b)
CMPSD for unordered	__m128d _mm_cmpunord_sd(__m128d a, __m128d b)
CMPSD for not-less-than-or-equal	__m128d _mm_cmpnle_sd(__m128d a, __m128d b)

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 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.

CMPSS—Compare Scalar Single-Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F C2 /r ib	CMPSS *xmm1, xmm2/m32, imm8*	Compare low single-precision floating-point value in *xmm2/m32* and *xmm1* using *imm8* 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 3-6); 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; the ordered relationship is true when neither source operand is a NaN

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, since a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Some of the comparisons listed in Table 3-6 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 3-6 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction. See Table 3-10

Table 3-10. Pseudo-Ops and CMPSS

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

The greater-than relations 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	__m128 _mm_cmpeq_ss(__m128 a, __m128 b)
CMPSS for less-than	__m128 _mm_cmplt_ss(__m128 a, __m128 b)
CMPSS for less-than-or-equal	__m128 _mm_cmple_ss(__m128 a, __m128 b)
CMPSS for greater-than	__m128 _mm_cmpgt_ss(__m128 a, __m128 b)
CMPSS for greater-than-or-equal	__m128 _mm_cmpge_ss(__m128 a, __m128 b)
CMPSS for inequality	__m128 _mm_cmpneq_ss(__m128 a, __m128 b)

CMPSS for not-less-than	__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)
CMPSS for not-greater-than	__m128 _mm_cmpngt_ss(__m128 a, __m128 b)
CMPSS for not-greater-than-or-equal	__m128 _mm_cmpnge_ss(__m128 a, __m128 b)
CMPSS for ordered	__m128 _mm_cmpord_ss(__m128 a, __m128 b)
CMPSS for unordered	__m128 _mm_cmpunord_ss(__m128 a, __m128 b)
CMPSS for not-less-than-or-equal	__m128 _mm_cmpnle_ss(__m128 a, __m128 b)

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 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 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.

CMPXCHG—Compare and Exchange

Opcode	Instruction	Description
0F B0/*r*	CMPXCHG *r/m8,r8*	Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL.
0F B1/*r*	CMPXCHG *r/m16,r16*	Compare AX with *r/m16*. If equal, ZF is set and *r16* is loaded into *r/m16*. Else, clear ZF and load *r/m16* into AX
0F B1/*r*	CMPXCHG *r/m32,r32*	Compare EAX with *r/m32*. If equal, ZF is set and *r32* is loaded into *r/m32*. Else, clear ZF and load *r/m32* into EAX

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.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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 *m64*	Compare EDX:EAX with *m64*. If equal, set ZF and load ECX:EBX into *m64*. Else, clear ZF and load *m64* 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.

Protected Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

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

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

Description

Compares the double-precision floating-point values in the low quadwords of 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 an 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)

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

GP(0)	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.

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

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

Description

Compares the single-precision floating-point values in the low doublewords 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 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals an 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)

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

GP(0)	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.

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

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.

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

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

Table 3-11 shows information returned, depending on the initial value loaded into the EAX register. Table 3-12 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a higher value entered than is valid 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. For a Pentium 4 processor, entering an input value of 80000005H or above returns the information for an input value of 2.

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

See also:

"Serializing Instructions" in Chapter 7 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*

AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618)

Table 3-11. 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 3-12) “Genu” “ntel” “inel”
01H	EAX EBX ECX EDX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-5) Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value ∗ 8 = cache line size in bytes) Bits 23-16: Number of logical processors per physical processor; two for the Pentium 4 processor supporting Hyper-Threading Technology Bits 31-24: Local APIC ID Extended Feature Information (see Figure 3-6 and Table 3-14) Feature Information (see Figure 3-7 and Table 3-15)
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 3-16) Cache and TLB Information Cache and TLB Information Cache and TLB Information
03H	EAX EBX ECX EDX 	Reserved. Reserved. Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) **NOTE:** Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature. See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) for more information on PSN.
04H	 EAX EBX ECX	*Deterministic Cache Parameters Leaf* Bits 4-0: Cache Type** Bits 7-5: Cache Level (starts at 1) Bits 8: Self Initializing cache level (does not need SW initialization) Bits 9: Fully Associative cache Bits 13-10: Reserved Bits 25-14: Number of threads sharing this cache* Bits 31-26: Number of processor cores on this die (Multicore)* Bits 11-00: L = System Coherency Line Size* Bits 21-12: P = Physical Line partitions* Bits 31-22: W = Ways of associativity* Bits 31-00: S = Number of Sets*

Table 3-11. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Reserved = 0 *Add one to the value in the register file to get the number. For example, the number of processor cores is EAX[31:26]+1. ** Cache Types fields 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved NOTE: CPUID leaves > 3 < 80000000 are only visible when IA32_CR_MISC_ENABLES.BOOT_NT4 (bit 22) is clear (Default)
5H		*MONITOR/MWAIT Leaf*
	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 3-12).
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
80000001H	EAX	Extended Processor Signature and Extended Feature Bits. (Currently Reserved
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
80000002H	EAX	Processor Brand String
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000003H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000004H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000005H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0

Table 3-11. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
80000006H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Bits 0-7: Cache Line Size Bits 15-12: L2 Associativity Bits 31-16: Cache size in 1K units
	EDX	Reserved = 0
80000007H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
80000008H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0

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

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 3-12) and is processor specific.

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

```
EBX ← 756e6547h (* "Genu", with G in the low 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 *)
```

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

When CPUID executes with EAX set to 0, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 3-12) and is processor specific.

Table 3-12. 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	01H	Not Implemented

Table 3-12. Highest CPUID Source Operand for IA-32 Processors (Contd.)

IA-32 Processors	Highest Value in EAX	
	Basic Information	Extended Function Information
Pentium Pro and Pentium II Processors, Intel® Celeron™ Processors	02H	Not Implemented
Pentium III Processors	03H	Not Implemented
Pentium 4 Processors	02H	80000004H
Intel Xeon Processors	02H	80000004H
Pentium M Processor	02H	80000004H
Pentium 4 Processor supporting Hyper-Threading Technology	05H	80000008H

Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*.

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

\When CPUID executes with EAX set to 1, version information is returned in EAX (see Figure 3-5). For example: model, family, and processor type for the first processor in the Intel Pentium 4 family is returned as follows:

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

See Table 3-13 for available processor type values. Stepping IDs are provided as needed.

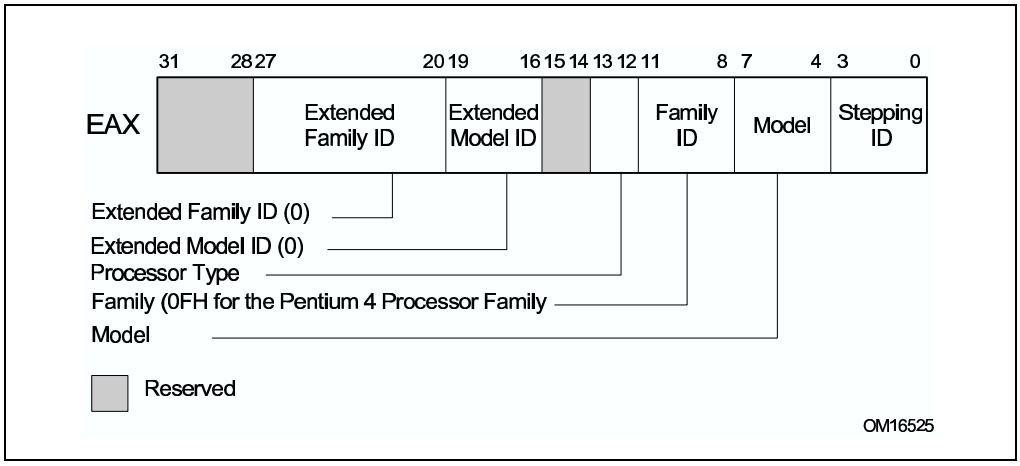

Figure 3-5. Version Information Returned by CPUID in EAX

Table 3-13. Processor Type Field

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

NOTE

See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) and Chapter 14 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID and Extended Model ID need be examined only if the Family ID reaches 0FH. Always display processor information as a combination of family, model, and stepping.

Integrate the ID fields into a display as:

```
Displayed family = ((Extended Family ID(4-bits) << 4)) (8-bits)
+ Family ID (4-bits zero extended to 8-bits)
```

Compute the displayed model from the Model ID and the Extended Model ID as:

```
Displayed Model = ((Extended Model ID (4-bits) << 4))(8-bits)
+ Model (4-bits zero extended to 8-bits)
```

INPUT EAX = 1: Returns Additional Information in EBX

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

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

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

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

- Figure 3-6 and Table 3-14 show encodings for ECX.
- Figure 3-7 and Table 3-15 show encodings for EDX.

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

NOTE

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

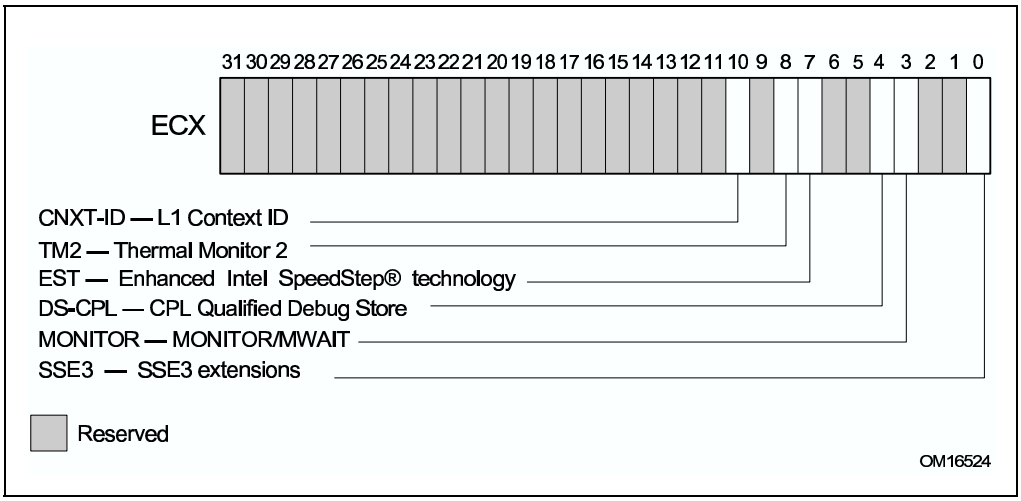

Figure 3-6. Extended Feature Information Returned in the ECX Register

Table 3-14. More on Extended Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3).** A value of 1 indicates the processor supports this technology.
3	MONITOR	**MONITOR/MWAIT.** A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store.** A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
7	EST	**Enhanced Intel SpeedStep® technology.** A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2.** A value of 1 indicates whether the processor supports this technology.
10	CNXT-ID	**L1 Context ID**. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.

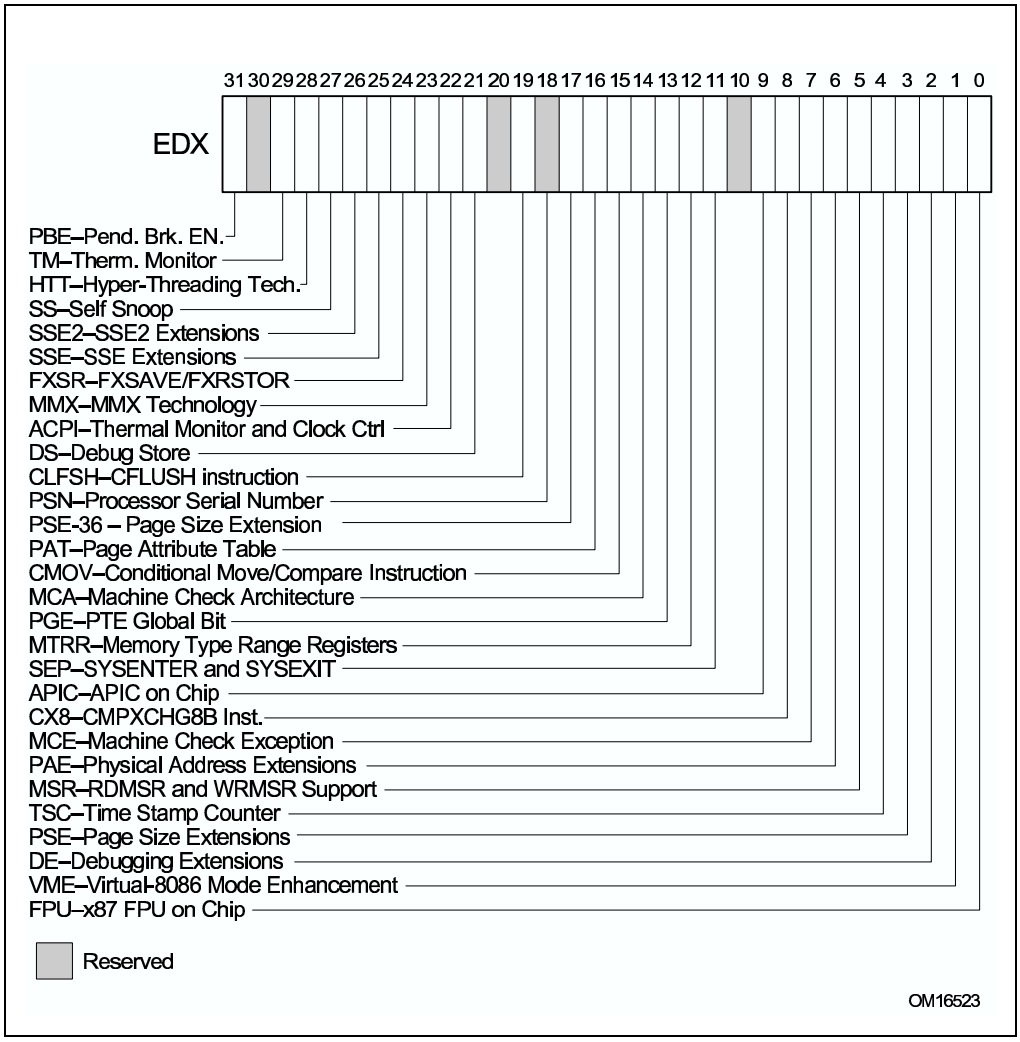

Figure 3-7. Feature Information Returned in the EDX Register

Table 3-15. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	**Floating Point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.

Table 3-15. More on Feature Information Returned in the EDX Register (Contd.)

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

Table 3-15. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
17	PSE-36	**36-Bit Page Size Extension.** Extended 4-MByte pages that are capable of addressing physical memory beyond 4 GBytes are supported. This feature indicates that the upper four bits of the physical address of the 4-MByte page is encoded by bits 13-16 of the page directory entry.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	Reserved	Reserved
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 15, *Debugging and Performance Monitoring*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*).
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.

Table 3-15. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
28	HTT	**Hyper-Threading Technology.** The processor supports Hyper-Threading Technology.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

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

When CPUID executes with EAX set to 2, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers.

The encoding is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs. The first member of the family of Pentium 4 processors will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 3-16 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 3-16. Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4 KByte Pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4 MByte Pages, 4-way set associative, 2 entries
03H	Data TLB: 4KByte Pages, 4-way set associative, 64 entries
04H	Data TLB: 4MByte Pages, 4-way set associative, 8 entries
06H	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
0AH	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0CH	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
22H	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
25H	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	3rd-level cache: 4M Bytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	1st-level data cache: 32K Bytes, 8-way set associative, 64 byte line size
30H	1st-level instruction cache: 32K Bytes, 8-way set associative, 64 byte line size
40H	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
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
5BH	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
66H	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size

Table 3-16. Encoding of Cache and TLB Descriptors (Contd.)

Descriptor Value	Cache or TLB Description
68H	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Trace cache: 12 K-µop, 8-way set associative
71H	Trace cache: 16 K-µop, 8-way set associative
72H	Trace cache: 32 K-µop, 8-way set associative
78H	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
82H	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
B0H	Instruction TLB: 4 KByte Pages, 4-way set associative, 128 entries
B3H	Data TLB: 4 KByte Pages, 4-way set associative, 128 entries
F0H	64-Byte Prefetching
F1H	128-Byte Prefetching

Example 3-1. Example of Cache and TLB Interpretation

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

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

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This indicates that CPUID needs to be executed once with an input value of 2 to retrieve complete information about caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - 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 sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

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

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 14 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*.

The Processor Brand String Method

Figure 3-8 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all IA-32 architecture compatible processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the maximum operating frequency of the processor to the EAX, EBX, ECX, and EDX registers.

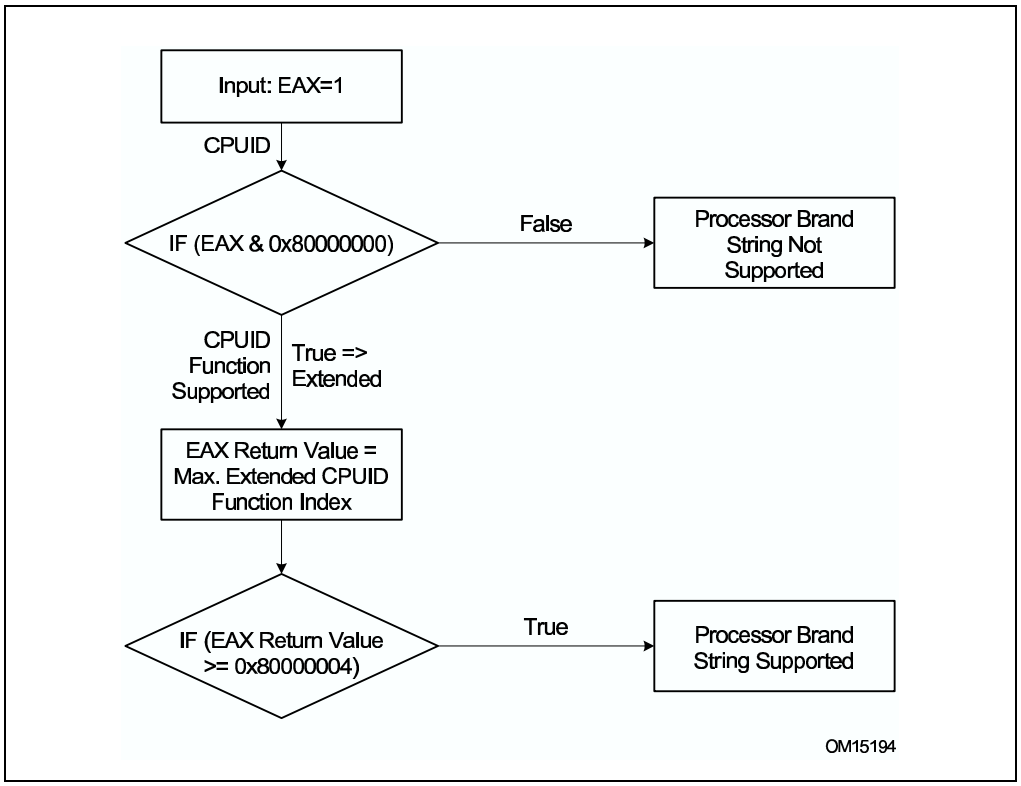

Figure 3-8. Determination of Support for the Processor Brand String

How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL terminated.

Table 3-17 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-17. Processor Brand String Returned with Pentium 4 Processor

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H	“ ” “ ” “ ” “nI ”

Table 3-17. Processor Brand String Returned with Pentium 4 Processor

80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	“(let” “P)R” “itne” “R(mu”
80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	“ 4)” “ UPC” “0051” “\0zHM”

Extracting the Maximum Processor Frequency from Brand Strings

Figure 3-9 provides an algorithm which software can use to extract the maximum processor operating frequency from the processor brand string.

NOTE

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

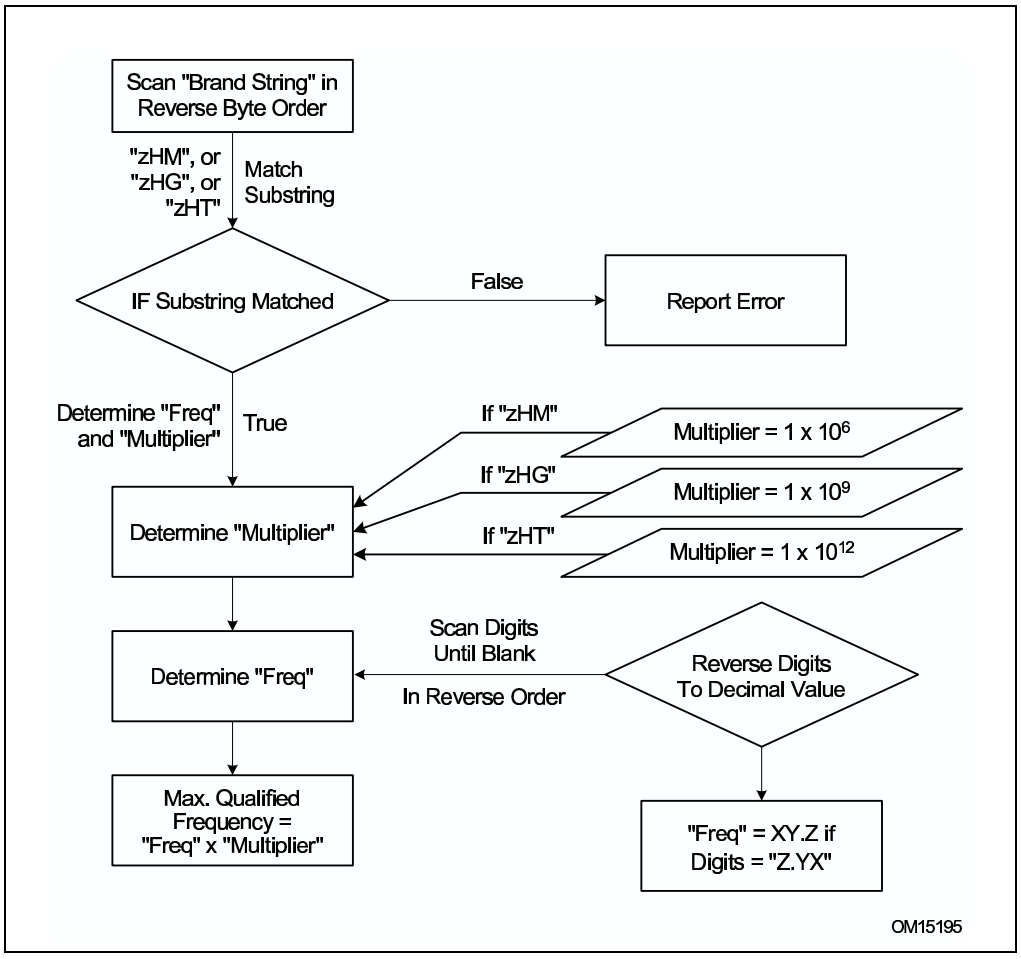

Figure 3-9. Algorithm for Extracting Maximum Processor Frequency

The Processor Brand Index Method

The brand index method (introduced with Pentium III Xeon processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associate with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

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

reserved, allowing for backward compatibility with processors that do not support the brand identification feature.

Table 3-18 shows brand indices that have identification strings associated with them.

Table 3-18. Mapping of Brand Indices and IA-32 Processor Brand Strings

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

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

IA-32 Architecture Compatibility

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

Operation

```
IA32_BIOS_SIGN_ID MSR ← Update with installed microcode revision number.
CASE (EAX) OF
    EAX = 0:
        EAX ← highest basic function input value understood by CPUID;
        EBX ← Vendor identification string;
        EDX ← Vendor identification string;
        ECX ← Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] ← Stepping ID;
        EAX[7:4] ← Model;
        EAX[11:8] ← Family;
        EAX[13:12] ← Processor type;
        EAX[15:14] ← Reserved;
        EAX[19:16] ← Extended Model;
        EAX[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 ← Feature flags; (* See Figure 3-6 *)
        EDX ← Feature flags;  (* See Figure 3-7 *)
    BREAK;
    EAX = 2H:
        EAX ← Cache and TLB information;
        EBX ← Cache and TLB information;
        ECX ← Cache and TLB information;
        EDX ← Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← ProcessorSerialNumber[31:0];
        (* Pentium III processors only, otherwise reserved *)
        EDX ← ProcessorSerialNumber[63:32];
        (* Pentium III processors only, otherwise reserved *
    BREAK
    EAX = 4H:
        EAX ← Deterministic Cache Parameters Leaf; /* see page 3-121 */
        EBX ← Deterministic Cache Parameters Leaf;
        ECX ← Deterministic Cache Parameters Leaf;
        EDX ← Deterministic Cache Parameters Leaf;
    BREAK;
    EAX = 5H:
        EAX ← MONITOR/MWAIT Leaf; /* see page 3-122 */
```

```
        EBX ← MONITOR/MWAIT Leaf;
        ECX ← MONITOR/MWAIT Leaf;
        EDX ← MONITOR/MWAIT Leaf;
    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 Brand String;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000003H:
        EAX ← Processor Brand String, continued;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000004H:
        EAX ← Processor Brand String, continued;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000005H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000006H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Cache information;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000007H:
```

```
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000008H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    DEFAULT: (* EAX > 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 *xmm1, xmm2/m64*	Convert two packed signed doubleword integers from *xmm2/m128* to two packed double-precision floating-point values in *xmm1*.

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.
#UD	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 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.
#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 *xmm1*, *xmm2/m128*	Convert four packed signed doubleword integers from *xmm2/m128* to four packed single-precision floating-point values in *xmm1*.

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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *xmm1, xmm2/m128*	Convert two packed double-precision floating-point values from *xmm2/m128* to two packed signed doubleword integers in *xmm1*.

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand and the high quadword is cleared to all 0s.

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

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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *mm, xmm/m128*	Convert two packed double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm*.

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

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

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

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 a 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.
#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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 5A /r	CVTPD2PS *xmm1, xmm2/m128*	Convert two packed double-precision floating-point values in *xmm2/m128* to two packed single-precision floating-point values in *xmm1*.

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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *xmm, mm/m64*	Convert two packed signed doubleword integers from *mm/mem64* to two packed double-precision floating-point values in *xmm*.

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand). The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register.

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.
#UD	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 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 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.

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

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

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand). The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

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.

#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

GP(0)	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 *xmm1*, *xmm2/m128*	Convert four packed single-precision floating-point values from *xmm2/m128* to four packed signed doubleword integers in *xmm1*.

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 floating-point invalid exception is raised, and if this exception is masked, 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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.

CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
0F 5A /r	CVTPS2PD *xmm1*, *xmm2/m64*	Convert two packed single-precision floating-point values in *xmm2/m64* to two packed double-precision floating-point values in *xmm1*.

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

CVTPS2PD __m128d _mm_cvtps_pd(__m128 a)

SIMD Floating-Point Exceptions

Invalid, 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 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 *mm, xmm/m64*	Convert two packed single-precision floating-point values from *xmm/m64* to two packed signed doubleword integers in *mm*.

Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

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

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.

#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

GP(0)	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 *r32, xmm/m64*	Convert one double-precision floating-point value from *xmm/m64* to one signed doubleword integer *r32*.

Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

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

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.
	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 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 *xmm1, xmm2/m64*	Convert one double-precision floating-point value in *xmm2/m64* to one single-precision floating-point value in *xmm1*.

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.
	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 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 *xmm*, r/m32	Convert one signed doubleword integer from *r/m32* to one double-precision floating-point value in *xmm*.

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.

#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 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 *xmm, r/m32*	Convert one signed doubleword integer from *r/m32* to one single-precision floating-point value in *xmm*.

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.
	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 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 *xmm1*, *xmm2/m32*	Convert one single-precision floating-point value in *xmm2/m32* to one double-precision floating-point value in *xmm1*.

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_cvtss_sd(__m128d a, __m128 b)

SIMD Floating-Point Exceptions

Invalid, 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 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 *r32, xmm/m32*	Convert one single-precision floating-point value from *xmm/m32* to one signed doubleword integer in *r32*.

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 floating-point invalid exception is raised, and if this exception is masked, 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.
	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

GP(0)	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 *mm, xmm/m128*	Convert two packer double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm* 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 technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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.
	If a 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.
#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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *xmm1*, *xmm2/m128*	Convert two packed double-precision floating-point values from *xmm2/m128* to two packed signed doubleword integers in *xmm1* 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 floating-point invalid exception is raised, and if this exception is masked, 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *xmm1*, *xmm2/m128*	Convert four single-precision floating-point values from *xmm2/m128* to four signed doubleword integers in *xmm1* 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 floating-point invalid exception is raised, and if this exception is masked, 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *mm, xmm/m64*	Convert two single-precision floating-point values from *xmm/m64* to two signed doubleword signed integers in *mm* 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 technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

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.

#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

GP(0)	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 *r32, xmm/m64*	Convert one double-precision floating-point value from *xmm/m64* to one signed doubleword integer in *r32* 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 floating-point invalid exception is raised, and if this exception is masked, 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_cvttsd_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.
	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 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 *r32, xmm/m32*	Convert one single-precision floating-point value from *xmm/m32* to one signed doubleword integer in *r32* 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 floating-point invalid exception is raised, and if this exception is masked, 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_cvttss_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.
	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 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 the 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

```
old_AL ← AL;
old_CF ← CF;
CF ← 0;
IF (((AL AND 0FH) > 9) OR AF = 1)
  THEN
    AL ← AL + 6;
    CF ← old_CF OR (Carry from AL ← AL + 6);
    AF ← 1;
  ELSE
    AF ← 0;
FI;
IF ((old_AL > 99H) OR (old_CF = 1))
  THEN
    AL ← AL + 60H;
    CF ← 1;
  ELSE
    CF ← 0;
FI;
```

Example

```
ADD AL, BL        Before: AL=79H  BL=35H EFLAGS(OSZAPC)=XXXXXX
                  After:  AL=AEH  BL=35H EFLAGS(OSZAPC)=110000
DAA               Before: AL=AEH  BL=35H EFLAGS(OSZAPC)=110000
                  After:  AL=14H  BL=35H EFLAGS(OSZAPC)=X00111
DAA               Before: AL=2EH  BL=35H EFLAGS(OSZAPC)=110000
                  After:  AL=34H  BL=35H EFLAGS(OSZAPC)=X00101
```

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the "Operation" section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

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

```
old_AL ← AL;
old_CF ← CF;
CF ← 0;
IF (((AL AND 0FH) > 9) OR AF = 1)
  THEN
    AL ← AL – 6;
    CF ← old_CF OR (Borrow from AL ← AL – 6);
    AF ← 1;
  ELSE
    AF ← 0;
FI;
IF ((old_AL > 99H) OR (old_CF = 1))
  THEN
    AL ← AL – 60H;
    CF ← 1;
  ELSE
    CF ← 0;
FI;
```

Example

```
SUB AL, BL       Before: AL=35H  BL=47H EFLAGS(OSZAPC)=XXXXXX
                 After:  AL=EEH  BL=47H EFLAGS(0SZAPC)=010111
DAA              Before: AL=EEH  BL=47H EFLAGS(OSZAPC)=010111
                 After:  AL=88H  BL=47H EFLAGS(0SZAPC)=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 *r/m8*	Decrement *r/m8* by 1.
FF /1	DEC *r/m16*	Decrement *r/m16* by 1.
FF /1	DEC *r/m32*	Decrement *r/m32* by 1.
48+rw	DEC *r16*	Decrement *r16* by 1.
48+rd	DEC *r32*	Decrement *r32* 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 non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#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.

DIV—Unsigned Divide

Opcode	Instruction	Description
F6 /6	DIV *r/m8*	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
F7 /6	DIV *r/m16*	Unsigned divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /6	DIV *r/m32*	Unsigned divide EDX:EAX by *r/m32*, 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). See Table 3-19.

Table 3-19. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$

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;
    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.
#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 *xmm1, xmm2/m128*	Divide packed double-precision floating-point values in *xmm1* by packed double-precision floating-point values *xmm2/m128.*

Description

Performs an SIMD divide of the *four* packed double-precision floating-point values in the destination operand (first operand) by the *four* 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 an 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *xmm1, xmm2/m128*	Divide packed single-precision floating-point values in *xmm1* by packed single-precision floating-point values *xmm2/m128*.

Description

Performs an 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 an 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 a 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 *xmm1, xmm2/m64*	Divide low double-precision floating-point value n *xmm1* by low double-precision floating-point value in *xmm2/mem64.*

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.

#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 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 *xmm1*, *xmm2/m32*	Divide low single-precision floating-point value in *xmm1* by low single-precision floating-point value in *xmm2/m32*.

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.

#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 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 Technology 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 technology 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 technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

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 *iw* 00	ENTER *imm16*,0	Create a stack frame for a procedure.
C8 *iw* 01	ENTER *imm16*,1	Create a nested stack frame for a procedure.
C8 *iw* ib	ENTER *imm16,imm8*	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;
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.

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 $(2^{ST(0)} - 1)$.

Description

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range –1.0 to +1.0. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-20. Results Obtained from F2XM1

ST(0) SRC	ST(0) DEST
–1.0 to –0	–0.5 to –0
–0	–0
+0	+0
+0 to +1.0	+0 to 1.0

Values other than 2 can be exponentiated using the following formula:

$x^y \leftarrow 2^{(y * \log_2 x)}$

Operation

$ST(0) \leftarrow (2^{ST(0)} - 1)$;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source is a denormal value.
#U	Result is too small for destination format.

#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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.

Table 3-21. Results Obtained from FABS

ST(0) SRC	ST(0) DEST
–∞	+∞
–F	+F
–0	+0
+0	+0
+F	+F
+∞	+∞
NaN	NaN

NOTE:
F Means finite floating-point value.

Operation

ST(0) ← |ST(0)|

FPU Flags Affected

C1 Set to 0 if stack underflow occurred; otherwise, set 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.

Virtual-8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FADD/FADDP/FIADD—Add

Opcode	Instruction	Description
D8 /0	FADD *m32fp*	Add *m32fp* to ST(0) and store result in ST(0).
DC /0	FADD *m64fp*	Add *m64fp* 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 *m32int*	Add *m32int* to ST(0) and store result in ST(0).
DE /0	FIADD *m16int*	Add *m16int* to ST(0) and store result in ST(0).

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is –0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-22.

Table 3-22. FADD/FADDP/FIADD Results

		DEST						
		–∞	–F	–0	+0	+F	+∞	NaN
SRC	-∞	-∞	-∞	-∞	-∞	-∞	*	NaN
	–F or –I	-∞	–F	SRC	SRC	±F or ±0	+∞	NaN
	–0	-∞	DEST	–0	±0	DEST	+∞	NaN
	+0	-∞	DEST	±0	+0	DEST	+∞	NaN
	+F or +I	-∞	±F or ±0	SRC	SRC	+F	+∞	NaN
	+∞	*	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FIADD
    THEN
        DEST ← DEST + ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* source operand is floating-point value *)
        DEST ← DEST + SRC;
FI;
IF instruction = FADDP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of unlike sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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 *m80 dec*	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 ← TOP – 1;
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

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.

Table 3-23. FBSTP Results

ST(0)	DEST
-∞ or Value Too Large for DEST Format	*
F ≤ –1	–D
–1 < F < –0	**
–0	–0
+0	+0
+0 < F < +1	**
F ≥ +1	+D
+∞ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

D Means packed-BCD number.

* Indicates floating-point invalid-operation (#IA) exception.

** ±0 or ±1, depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

Operation

```
DEST ← BCD(ST(0));
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value that exceeds 18 BCD digits in length.
	Source operand is an SNaN, QNaN, ±∞, or in an unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a segment register is being loaded with a segment selector that points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

FCHS—Change Sign

Opcode	Instruction	Description
D9 E0	FCHS	Complements sign of ST(0)

Description

Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

Table 3-24. FCHS Results

ST(0) SRC	ST(0) DEST
–∞	+∞
–F	+F
–0	+0
+0	–0
+F	–F
+∞	–∞
NaN	NaN

NOTE:
F Means finite floating-point value.

Operation

SignBit(ST(0)) ← NOT (SignBit(ST(0)))

FPU Flags Affected

C1 Set to 0 if stack underflow occurred; otherwise, set 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.

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. See the IA-32 Architecture Compatibility section below.

Description

Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *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, Intel Xeon, or P6 family processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCRS register.

Operation

```
FPUStatusWord[0..7] ← 0;
FPUStatusWord[15] ← 0;
```

FPU Flags Affected

The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	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*cc*—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 condition for each mnemonic os 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*cc* instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOV*cc* instructions. Software can check if the FCMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS" in this chapter). If both the CMOV and FPU feature bits are set, the FCMOV*cc* instructions are supported.

IA-32 Architecture Compatibility

The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

```
IF condition TRUE
    ST(0) ← ST(i)
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Integer Flags Affected

None.

Protected Mode Exceptions

#NM	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 *m32fp*	Compare ST(0) with *m32fp*.
DC /2	FCOM *m64fp*	Compare ST(0) with *m64fp*.
D8 D0+i	FCOM ST(i)	Compare ST(0) with ST(i).
D8 D1	FCOM	Compare ST(0) with ST(1).
D8 /3	FCOMP *m32fp*	Compare ST(0) with *m32fp* and pop register stack.
DC /3	FCOMP *m64fp*	Compare ST(0) with m64fp 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 is equal to +0.0.

Table 3-25. FCOM/FCOMP/FCOMPP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered*	1	1	1

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.

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, set to 0.
C0, C2, C3	See table on previous page.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats.
	Register is marked empty.
#D	One or both operands are denormal values.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

Table 3-26. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results

Comparison Results	ZF	PF	CF
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered*	1	1	1

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 FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions clear the OF flag in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

```
CASE (relation of operands) OF
    ST(0) > ST(i):      ZF, PF, CF ← 000;
    ST(0) < ST(i):      ZF, PF, CF ← 001;
    ST(0) = ST(i):      ZF, PF, CF ← 100;
ESAC;
IF instruction is FCOMI or FCOMIP
    THEN
        IF ST(0) or ST(i) = NaN or unsupported format
            THEN
                #IA
                IF FPUControlWord.IM = 1
                    THEN
                        ZF, PF, CF ← 111;
                FI;
        FI;
FI;
IF instruction is FUCOMI or FUCOMIP
    THEN
        IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format
            THEN
                ZF, PF, CF ← 111;
            ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)
                 #IA;
                IF FPUControlWord.IM = 1
                    THEN
                        ZF, PF, CF ← 111;
                FI;
        FI;
FI;
IF instruction is FCOMIP or FUCOMIP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, set to 0.
C0, C2, C3	Not affected.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	(FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats. (FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.

Protected Mode Exceptions

#NM	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.

Table 3-27. FCOS Results

ST(0) SRC	ST(0) DEST
–∞	*
–F	–1 to +1
–0	+1
+0	+1
+F	–1 to +1
+∞	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π. 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 π in performing such reductions.

Operation

```
IF |ST(0)| < 2^63
THEN
    C2 ← 0;
    ST(0) ← cosine(ST(0));
ELSE (*source operand is out-of-range *)
    C2 ← 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
	Undefined if C2 is 1.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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. 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 *m32fp*	Divide ST(0) by *m32fp* and store result in ST(0).
DC /6	FDIV *m64fp*	Divide ST(0) by *m64fp* 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 *m32int*	Divide ST(0) by *m32int* and store result in ST(0).
DE /6	FIDIV *m16int*	Divide ST(0) by *m64int* and store result in ST(0).

Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-28. FDIV/FDIVP/FIDIV Results

		DEST						
		−∞	−F	−0	+0	+F	+∞	NaN
SRC	-∞	*	+0	+0	−0	−0	*	NaN
	−F	+∞	+F	+0	−0	−F	−∞	NaN
	−I	+∞	+F	+0	−0	−F	−∞	NaN
	−0	+∞	**	*	*	**	−∞	NaN
	+0	−∞	**	*	*	**	+∞	NaN
	+I	−∞	−F	−0	+0	+F	+∞	NaN
	+F	−∞	−F	−0	+0	+F	+∞	NaN
	+∞	*	−0	−0	+0	+0	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

Operation

```
IF SRC = 0
    THEN
        #Z
    ELSE
        IF instruction is FIDIV
            THEN
                DEST ← DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
            ELSE (* source operand is floating-point value *)
                DEST ← DEST / SRC;
        FI;
FI;
IF instruction = FDIVP
    THEN
        PopRegisterStack
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	$\pm\infty$ / $\pm\infty$; ±0 / ±0
#D	Source is a denormal value.
#Z	DEST / ±0, where DEST is not equal to ±0.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	Description
D8 /7	FDIVR *m32fp*	Divide *m32fp* by ST(0) and store result in ST(0)
DC /7	FDIVR *m64fp*	Divide *m64fp* 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 *m32int*	Divide *m32int* by ST(0) and store result in ST(0)
DE /7	FIDIVR *m16int*	Divide *m16int* by ST(0) and store result in ST(0)

Description

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version, divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-29. FDIVR/FDIVRP/FIDIVR Results

		DEST						
		–∞	–F	–0	+0	+F	+∞	NaN
SRC	–∞	*	+∞	+∞	–∞	–∞	*	NaN
	–F	+0	+F	**	**	-F	–0	NaN
	–I	+0	+F	**	**	-F	–0	NaN
	–0	+0	+0	*	*	–0	–0	NaN
	+0	–0	–0	*	*	+0	+0	NaN
	+I	–0	-F	**	**	+F	+0	NaN
	+F	–0	-F	**	**	+F	+0	NaN
	+∞	*	–∞	–∞	+∞	+∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0.

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;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	±∞ / ±∞; ±0 / ±0
#D	Source is a denormal value.
#Z	SRC / ±0, where SRC is not equal to ±0.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

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 *m16int*	Compare ST(0) with *m16int.*
DA /2	FICOM *m32int*	Compare ST(0) with *m32int.*
DE /3	FICOMP *m16int*	Compare ST(0) with *m16int* and pop stack register.
DA /3	FICOMP *m32int*	Compare ST(0) with *m32int* and pop stack register.

Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

Table 3-30. FICOM/FICOMP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered	1	1	1

These instructions perform an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine" in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to "unordered."

The sign of zero is ignored, so that –0.0 ← +0.0.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

Operation

```
CASE (relation of operands) OF
    ST(0) > SRC:    C3, C2, C0 ← 000;
    ST(0) < SRC:    C3, C2, C0 ← 001;
    ST(0) = SRC:    C3, C2, C0 ← 100;
    Unordered:      C3, C2, C0 ← 111;
ESAC;
IF instruction = FICOMP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 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 *m16int*	Push *m16int* onto the FPU register stack.
DB /0	FILD *m32int*	Push *m32int* onto the FPU register stack.
DF /5	FILD *m64int*	Push *m64int* onto the FPU register stack.

Description

Converts the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

Operation

```
TOP ← TOP – 1;
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

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

Description

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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 4, Intel Xeon, or P6 family processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers.

This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

Operation

```
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

C0, C1, C2, C3 set to 0.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	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 *m16int*	Store ST(0) in *m16int.*
DB /2	FIST *m32int*	Store ST(0) in *m32int.*
DF /3	FISTP *m16int*	Store ST(0) in *m16int* and pop register stack.
DB /3	FISTP *m32int*	Store ST(0) in *m32int* and pop register stack.
DF /7	FISTP *m64int*	Store ST(0) in *m64int* and pop register stack.

Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or doubleword integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-31. FIST/FISTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	–I
$-1 < F < -0$	**
–0	0
+0	0
$+0 < F < +1$	**
$F \geq +1$	+I
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

** 0 or ±1, depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

Operation

```
DEST ← Integer(ST(0));
IF instruction = FISTP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction of if the inexact exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
	Set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value is too large for the destination format.
	Source operand is an SNaN, QNaN, ±∞, or unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

FISTTP: Store Integer with Truncation

Opcode	Instruction	Description
DF /1	FISTTP *m16int*	Store ST as a signed integer (truncate) in *m16int* and pop ST.
DB /1	FISTTP *m32int*	Store ST as a signed integer (truncate) in *m32int* and pop ST.
DD /1	FISTTP *m64int*	Store ST as a signed integer (truncate) in *m64int* and pop ST.

Description

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-32. FISTTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	$-I$
$-1 < F < +1$	0
$F \geq +1$	$+I$
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

Notes:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

Operation

```
DEST ← ST;
pop ST;
```

Flags Affected

C1 is cleared; C0, C2, C3 undefined.

Numeric Exceptions

Invalid, Stack Invalid (stack underflow), Precision.

Protected Mode Exceptions

#GP(0)	If the destination is in a nonwritable segment.
	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#NM	If CR0.EM = 1.
	If TS bit in CR0 is set.
#UD	If CPUID.SSE3(ECX bit 0) = 0.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.EM = 1.
	If TS bit in CR0 is set.
#UD	If CPUID.SSE3(ECX bit 0) = 0.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.EM = 1.
	If TS bit in CR0 is set.
#UD	If CPUID.SSE3(ECX bit 0) = 0.
#PF(fault-code)	For a page fault.
#AC(0)	For unaligned memory reference if the current privilege is 3.

FLD—Load Floating Point Value

Opcode	Instruction	Description
D9 /0	FLD *m32fp*	Push *m32fp* onto the FPU register stack.
DD /0	FLD *m64fp*	Push *m64fp* onto the FPU register stack.
DB /5	FLD *m80fp*	Push *m80fp* 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, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.
#IA	Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD m80fp or FLD ST(i)).

Protected Mode Exceptions

#D	Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.
#GP(0)	If destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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 π onto the FPU register stack.
D9 EC	FLDLG2	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Push $\log_e 2$ 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, $\log_2 10$, $\log_2 e$, π, $\log_{10} 2$, and $\log_e 2$. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled "Pi" in Chapter 8 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for a description of the π constant.

Operation

```
TOP ← TOP – 1;
ST(0) ← CONSTANT;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#NM	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.

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 m2byte	Load FPU control word from *m2byte.*

Description

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU's mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *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.

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 *m14/28byte*	Load FPU environment from *m14byte* or *m28byte.*

Description

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *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.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

Operation

```
FPUControlWord ← SRC[FPUControlWord);
FPUStatusWord ← SRC[FPUStatusWord);
FPUTagWord ← SRC[FPUTagWord);
FPUDataPointer ← SRC[FPUDataPointer);
FPUInstructionPointer ← SRC[FPUInstructionPointer);
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode);
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next "waiting" floating-point instruction.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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 *m32fp*	Multiply ST(0) by *m32fp* and store result in ST(0)
DC /1	FMUL *m64fp*	Multiply ST(0) by *m64fp* 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 *m32int*	Multiply ST(0) by *m32int* and store result in ST(0)
DE /1	FIMUL *m16int*	Multiply ST(0) by *m16int* and store result in ST(0)

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating point or an integer value) and stores the product in the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or ∞. When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-33. FMUL/FMULP/FIMUL Results

		DEST						
		−∞	−F	−0	+0	+F	+∞	NaN
SRC	−∞	+∞	+∞	*	*	−∞	−∞	NaN
	−F	+∞	+F	+0	−0	−F	−∞	NaN
	−I	+∞	+F	+0	−0	−F	−∞	NaN
	−0	*	+0	+0	−0	−0	*	NaN
	+0	*	−0	−0	+0	+0	*	NaN
	+I	−∞	−F	−0	+0	+F	+∞	NaN
	+F	−∞	−F	−0	+0	+F	+∞	NaN
	+∞	−∞	−∞	*	*	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means Integer.

* Indicates invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FIMUL
    THEN
        DEST ← DEST * ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* source operand is floating-point value *)
        DEST ← DEST * SRC;
FI;
IF instruction = FMULP
    THEN
        PopRegisterStack
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	One operand is ±0 and the other is ±∞.

#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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(ST(1)/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 +π.

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point (–X,Y) is in the second quadrant, resulting in an angle between π/2 and π, while a point (X,–Y) is in the fourth quadrant, resulting in an angle between 0 and –π/2. A point (–X,–Y) is in the third quadrant, giving an angle between –π/2 and –π.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

Table 3-34. FPATAN Results

		ST(0)						
		–∞	–F	–0	+0	+F	+∞	NaN
ST(1)	–∞	–3π/4*	–π/2	–π/2	–π/2	–π/2	–π/4*	NaN
	–F	–π	–π to –π/2	–π/2	–π/2	–π/2 to –0	-0	NaN
	–0	–π	–π	–π*	–0*	–0	–0	NaN
	+0	+π	+π	+π*	+0*	+0	+0	NaN
	+F	+π	+π to +π/2	+π/2	+π/2	+π/2 to +0	+0	NaN
	+∞	+3π/4*	+π/2	+π/2	+π/2	+π/2	+π/4*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Table 8-10 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, specifies that the ratios 0/0 and ∞/∞ generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the 0/0 or ∞/∞ value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation, arctangent(0,0) etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

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) ← arctan(ST(1) / ST(0));
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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:

Remainder ← ST(0) – (Q ∗ ST(1))

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of [ST(0) / ST(1)] toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-35. FPREM Results

		ST(1)						
		–∞	–F	–0	+0	+F	+∞	NaN
ST(0)	–∞	*	*	*	*	*	*	NaN
	–F	ST(0)	–F or –0	**	**	–F or –0	ST(0)	NaN
	–0	–0	–0	*	*	–0	–0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	+F or +0	**	**	+F or +0	ST(0)	NaN
	+∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name "partial remainder" because of the way it computes the remainder. This 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.
#IA	Source operand is an SNaN value, modulus is 0, dividend is ∞, or unsupported format.

#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	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:

Remainder ← ST(0) – (Q * ST(1))

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of [ST(0) / ST(1)] toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-36. FPREM1 Results

		ST(1)						
		–∞	–F	–0	+0	+F	+∞	NaN
ST(0)	–∞	*	*	*	*	*	*	NaN
	–F	ST(0)	±F or –0	**	**	±F or –0	ST(0)	NaN
	–0	–0	–0	*	*	–0	–0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	±F or +0	**	**	±F or +0	ST(0)	NaN
	+∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the "Operation" section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of $\pi/4$), because it locates the original angle in the correct one of eight sectors of the unit circle.

Operation

```
D ← exponent(ST(0)) – exponent(ST(1));
IF D < 64
    THEN
        Q ← Integer(RoundTowardNearestInteger(ST(0) / ST(1)));
        ST(0) ← ST(0) – (ST(1) * Q);
        C2 ← 0;
        C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ← 1;
        N ← an implementation-dependent number between 32 and 63;
        QQ ← Integer(TruncateTowardZero((ST(0)  / ST(1)) / 2^(D – N)));
        ST(0) ← ST(0) – (ST(1) * QQ * 2^(D – N));
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS	Stack underflow occurred.

#IA	Source operand is an SNaN value, modulus (divisor) is 0, dividend is ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	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	Description
D9 F2	FPTAN	Replace ST(0) with its tangent and push 1 onto the FPU stack.

Description

Computes the tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than $\pm 2^{63}$. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

Table 3-37. FPTAN Results

ST(0) SRC	ST(0) DEST
$-\infty$	*
–F	–F to +F
–0	–0
+0	+0
+F	–F to +F
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π. 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 π in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

Operation

```
IF ST(0) < 2^63
THEN
    C2 ← 0;
    ST(0) ← tan(ST(0));
    TOP ← TOP – 1;
    ST(0) ← 1.0;
ELSE (*source operand is out-of-range *)
    C2 ← 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
	Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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 ∞, 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) ← RoundToIntegralValue(ST(0));
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#P	Source operand is not an integral value.

Protected Mode Exceptions

#NM	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 *m94/108byte*	Load FPU state from *m94byte* or *m108byte*.

Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *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 following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

Operation

```
FPUControlWord ← SRC[FPUControlWord);
FPUStatusWord ← SRC[FPUStatusWord);
FPUTagWord ← SRC[FPUTagWord);
FPUDataPointer ← SRC[FPUDataPointer);
FPUInstructionPointer ← SRC[FPUInstructionPointer);
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode);
ST(0) ← SRC[ST(0));
ST(1) ← SRC[ST(1));
ST(2) ← SRC[ST(2));
ST(3) ← SRC[ST(3));
ST(4) ← SRC[ST(4));
ST(5) ← SRC[ST(5));
ST(6) ← SRC[ST(6));
ST(7) ← SRC[ST(7));
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	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 *m94/108byte*	Store FPU state to *m94byte* or *m108byte* after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE *m94/108byte*	Store FPU environment to *m94byte* or *m108byte* without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU. See the IA-32 Architecture Compatibility section below.

Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *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 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.

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 4, Intel Xeon, or P6 family processor.

Operation

```
(* Save FPU State and Registers *)
DEST[FPUControlWord) ← FPUControlWord;
DEST[FPUStatusWord) ← FPUStatusWord;
DEST[FPUTagWord) ← FPUTagWord;
DEST[FPUDataPointer) ← FPUDataPointer;
DEST[FPUInstructionPointer) ← FPUInstructionPointer;
DEST[FPULastInstructionOpcode) ← FPULastInstructionOpcode;
DEST[ST(0)) ← ST(0);
DEST[ST(1)) ← ST(1);
DEST[ST(2)) ← ST(2);
DEST[ST(3)) ← ST(3);
DEST[ST(4)) ← ST(4);
DEST[ST(5)) ← ST(5);
DEST[ST(6)) ← ST(6);
DEST[ST(7)) ← ST(7);
(* Initialize FPU *)
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#NM	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

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-38. FSCALE Results

		ST(1)							
			–∞	–F	–0	+0	+F	+∞	NaN
ST(0)	–∞	NaN	–∞	–∞	–∞	–∞	–∞	–∞	NaN
	–F	–0	–F	–F	–F	–F	–F	–∞	NaN
	–0	–0	–0	–0	–0	–0	–0	NaN	NaN
	+0	+0	+0	+0	+0	+0	+0	NaN	NaN
	+F	+0	+F	+F	+F	+F	+F	+∞	NaN
	+∞	NaN	+∞	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the

FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

Operation

$ST(0) \leftarrow ST(0) * 2^{RoundTowardZero(ST(1))}$;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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.

Table 3-39. FSIN Results

SRC (ST(0))	DEST (ST(0))
–∞	*
–F	–1 to +1
–0	–0
+0	+0
+F	–1 to +1
+∞	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π. 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 π in performing such reductions.

Operation

```
IF ST(0) < 2^63
THEN
    C2 ← 0;
    ST(0) ← sin(ST(0));
ELSE (* source operand out of range *)
    C2 ← 1;
FI:
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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.

Table 3-40. FSINCOS Results

SRC	DEST	
ST(0)	ST(1) Cosine	ST(0) Sine
–∞	*	*
–F	–1 to +1	–1 to +1
–0	+1	–0
+0	+1	+0
+F	–1 to +1	–1 to +1
+∞	*	*
NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π. 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 π in performing such reductions.

Operation

```
IF ST(0) < 2^63
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.
	Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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.

Table 3-41. FSQRT Results

SRC (ST(0))	DEST (ST(0))
–∞	*
–F	*
–0	–0
+0	+0
+F	+F
+∞	+∞
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

ST(0) ← SquareRoot(ST(0));

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
	Source operand is a negative value (except for –0).

#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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 *m32fp*	Copy ST(0) to *m32fp*.
DD /2	FST *m64fp*	Copy ST(0) to *m64fp*.
DD D0+i	FST ST(i)	Copy ST(0) to ST(i).
D9 /3	FSTP *m32fp*	Copy ST(0) to *m32fp* and pop register stack.
DD /3	FSTP *m64fp*	Copy ST(0) to *m64fp* and pop register stack.
DB /7	FSTP *m80fp*	Copy ST(0) to *m80fp* 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 the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ±0, ±∞, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value's identity as a 0, ∞, or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

Operation

```
DEST ← ST(0);
IF instruction = FSTP
    THEN
        PopRegisterStack; FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format. Does not occur if the source operand is in double extended-precision floating-point format.
#U	Result is too small for the destination format.
#O	Result is too large for the destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

FSTCW/FNSTCW—Store x87 FPU Control Word

Opcode	Instruction	Description
9B D9 /7	FSTCW *m2byte*	Store FPU control word to *m2byte* after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW *m2byte*	Store FPU control word to *m2byte* without checking for pending unmasked floating-point exceptions. See IA-32 Architecture Compatibility section below.

NOTE:
*

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 4, Intel Xeon, or P6 family processor.

Operation

```
DEST ← FPUControlWord;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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 *m14/28byte*	Store FPU environment to *m14byte* or *m28byte* after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV* *m14/28byte*	Store FPU environment to *m14byte* or *m28byte* without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions. See the IA-32 Architecture Compatibility section below.

Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *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 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 4, Intel Xeon, or P6 family processor.

Operation

```
DEST[FPUControlWord) ← FPUControlWord;
DEST[FPUStatusWord) ← FPUStatusWord;
DEST[FPUTagWord) ← FPUTagWord;
DEST[FPUDataPointer) ← FPUDataPointer;
DEST[FPUInstructionPointer) ← FPUInstructionPointer;
DEST[FPULastInstructionOpcode) ← FPULastInstructionOpcode;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

FSTSW/FNSTSW—Store x87 FPU Status Word

Opcode	Instruction	Description
9B DD /7	FSTSW *m2byte*	Store FPU status word at *m2byte* 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* *m2byte*	Store FPU status word at *m2byte* 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 section below.

Description

Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled "Branching and Conditional Moves on FPU Condition Codes" in Chapter 8 of the *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 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, Intel Xeon, or P6 family processor.

Operation

```
DEST ← FPUStatusWord;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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.

FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	Description
D8 /4	FSUB *m32fp*	Subtract *m32fp* from ST(0) and store result in ST(0).
DC /4	FSUB *m64fp*	Subtract *m64fp* 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 *m32int*	Subtract *m32int* from ST(0) and store result in ST(0).
DE /4	FISUB *m16int*	Subtract *m16int* from ST(0) and store result in ST(0).

Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-42 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST – SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward –∞ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-42. FSUB/FSUBP/FISUB Results

		SRC						
		–∞	–F or –I	–0	+0	+F or +I	+∞	NaN
DEST	–∞	*	–∞	–∞	–∞	–∞	–∞	NaN
	–F	+∞	±F or ±0	DEST	DEST	–F	–∞	NaN
	–0	+∞	–SRC	±0	–0	–SRC	–∞	NaN
	+0	+∞	–SRC	+0	±0	–SRC	–∞	NaN
	+F	+∞	+F	DEST	DEST	±F or ±0	–∞	NaN
	+∞	+∞	+∞	+∞	+∞	+∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FISUB
    THEN
        DEST ← DEST – ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* source operand is floating-point value *)
        DEST ← DEST – SRC;
FI;
IF instruction is FSUBP
    THEN
        PopRegisterStack
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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 *m32fp*	Subtract ST(0) from *m32fp* and store result in ST(0).
DC /5	FSUBR *m64fp*	Subtract ST(0) from *m64fp* 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 *m32int*	Subtract ST(0) from *m32int* and store result in ST(0).
DE /5	FISUBR *m16int*	Subtract ST(0) from *m16int* and store result in ST(0).

Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward –∞ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-43. FSUBR/FSUBRP/FISUBR Results

		SRC						
		−∞	−F or −I	−0	+0	+F or +I	+∞	NaN
DEST	−∞	*	+∞	+∞	+∞	+∞	+∞	NaN
	−F	−∞	±F or ±0	−DEST	−DEST	+F	+∞	NaN
	−0	−∞	SRC	±0	+0	SRC	+∞	NaN
	+0	−∞	SRC	−0	±0	SRC	+∞	NaN
	+F	−∞	−F	−DEST	−DEST	±F or ±0	+∞	NaN
	+∞	−∞	−∞	−∞	−∞	−∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FISUBR
    THEN
        DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) – DEST;
    ELSE (* source operand is floating-point value *)
        DEST ← SRC – DEST;
FI;
IF instruction = FSUBRP
    THEN
        PopRegisterStack
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	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).

Table 3-44. FTST Results

Condition	C3	C2	C0
ST(0) > 0.0	0	0	0
ST(0) < 0.0	0	0	1
ST(0) = 0.0	1	0	0
Unordered	1	1	1

This instruction performs an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine" in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to "unordered" and the invalid operation exception is generated.

The sign of zero is ignored, so that –0.0 ← +0.0.

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, set to 0.
C0, C2, C3	See Table 3-44.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	The source operand is a NaN value or is in an unsupported format.
#D	The source operand is a denormal value.

Protected Mode Exceptions

#NM	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

Opcode	Instruction	Description
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 is equal to +0.0.

Table 3-45. FUCOM/FUCOMP/FUCOMPP Results

Comparison Results	C3	C2	C0
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered	1	1	1

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/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

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 3-45.

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.

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).

Table 3-46. FXAM Results

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

Operation

```
C1 ← sign bit of ST; (* 0 for positive, 1 for negative *)
CASE (class of value or number in ST(0)) OF
    Unsupported:C3, C2, C0 ← 000;
    NaN:        C3, C2, C0 ← 001;
    Normal:     C3, C2, C0 ← 010;
    Infinity:   C3, C2, C0 ← 011;
    Zero:       C3, C2, C0 ← 100;
    Empty:      C3, C2, C0 ← 101;
    Denormal:   C3, C2, C0 ← 110;
ESAC;
```

FPU Flags Affected

C1	Sign of value in ST(0).
C0, C2, C3	See Table 3-46.

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 ← ST(0);
        ST(0) ← SRC;
        SRC ← temp;
    ELSE
        temp ← ST(0);
        ST(0) ← ST(1);
        ST(1) ← temp;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, set 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.

Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FXRSTOR—Restore x87 FPU, MMX Technology, SSE, and SSE2 State

Opcode	Instruction	Description
0F AE /1	FXRSTOR *m512byte*	Restore the x87 FPU, MMX technology, XMM, and MXCSR register state from *m512byte*.

Description

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and the first byte of the data should be located on a 16-byte boundary. Table 3-47 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 3-47. 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 an 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See 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.
#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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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.

FXSAVE—Save x87 FPU, MMX Technology, SSE, and SSE2 State

Opcode	Instruction	Description
0F AE /0	FXSAVE *m512byte*	Save the x87 FPU, MMX technology, XMM, and MXCSR register state to *m512byte*.

Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. Table 3-47 shows the layout of the state information in memory.

Table 3-47. 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

The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-47 are as follows:

FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for the layout of the x87 FPU opcode field.
FPU IP	x87 FPU Instruction Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed: • 32-bit mode—32-bit IP offset. • 16-bit mode—low 16 bits are IP offset; high 16 bits are reserved. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for a description of the x87 FPU instruction pointer.
CS	x87 FPU Instruction Pointer Selector (16 bits).

FPU DP	x87 FPU Instruction Operand (Data) Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed: • 32-bit mode—32-bit IP offset. • 16-bit mode—low 16 bits are IP offset; high 16 bits are reserved. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for a description of the x87 FPU operand pointer.
DS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits).
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent.
MXCSR_ MASK	MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See "Guidelines for Writing to the MXCSR Register" in Chapter 11 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for instructions for how to determine and use the MXCSR_MASK value.
ST0/MM0 through ST7/MM7	x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved.
XMM0 through XMM7	XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent.

The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

R7	R6	R5	R4	R3	R2	R1	R0
11	xx	xx	xx	11	11	11	11

Here, 11B indicates empty stack elements and "xx" indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

R7	R6	R5	R4	R3	R2	R1	R0
0	1	1	1	0	0	0	0

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be used by an application program to pass a "clean" x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.
- The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management). This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

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 technology registers) using Table 3-48:

Table 3-48. Recreating FSAVE Format

Exponent all 1's	Exponent all 0's	Fraction all 0's	J and M bits	FTW valid bit	x87 FTW	
0	0	0	0x	1	Special	10
0	0	0	1x	1	Valid	00
0	0	1	00	1	Special	10
0	0	1	10	1	Valid	00
0	1	0	0x	1	Special	10
0	1	0	1x	1	Special	10
0	1	1	00	1	Zero	01
0	1	1	10	1	Special	10
1	0	0	1x	1	Special	10
1	0	0	1x	1	Special	10
1	0	1	00	1	Special	10
1	0	1	10	1	Special	10
For all legal combinations above				0	Empty	11

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

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 a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See the description of the alignment check exception [#AC] below.)
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If 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 override 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 a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If 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 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.

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 logb(*x*) function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of $-\infty$ is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

Operation

```
TEMP ← Significand(ST(0));
ST(0) ← Exponent(ST(0));
TOP← TOP – 1;
ST(0) ← TEMP;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#Z	ST(0) operand is ±0.
#D	Source operand is a denormal value.

Protected Mode Exceptions

#NM	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 resister ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-49. FYL2X Results

		ST(0)							
		−∞	−F	±0	+0 < +F < +1	+1	+F > +1	+∞	NaN
ST(1)	−∞	*	*	+∞	+∞	*	−∞	−∞	NaN
	−F	*	*	**	+F	−0	−F	−∞	NaN
	−0	*	*	*	+0	−0	−0	*	NaN
	+0	*	*	*	−0	+0	+0	*	NaN
	+F	*	*	**	−F	+0	+F	+∞	NaN
	+∞	*	*	−∞	−∞	*	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains ±0, the instruction returns ∞ with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$\log_b x \leftarrow (\log_2 b)^{-1} * \log_2 x$

Operation

```
ST(1) ← ST(1) ∗ log2ST(0);
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN or unsupported format.
	Source operand in register ST(0) is a negative finite value (not –0).
#Z	Source operand in register ST(0) is ±0.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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 (ST(1) $*$ $\log_2$(ST(0) + 1.0)), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1-\sqrt{2}/2))\text{to}(1-\sqrt{2}/2)$$

The source operand in ST(1) can range from –∞ to +∞. If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

Table 3-50. FYL2XP1 Results

		ST(0)				
		–(1 – ($\sqrt{2}/2$)) to –0	–0	+0	+0 to +(1 – ($\sqrt{2}/2$))	NaN
ST(1)	–∞	+∞	*	*	–∞	NaN
	–F	+F	+0	–0	–F	NaN
	–0	+0	+0	–0	–0	NaN
	+0	–0	–0	+0	+0	NaN
	+F	–F	–0	+0	+F	NaN
	+∞	–∞	*	*	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon (ε) values, more significant digits can be retained by using the FYL2XP1 instruction than by using (ε+1) as an argument to the FYL2X instruction. The (ε+1) expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result of the FYL2XP1 instruction:

scale factor $\leftarrow \log_n 2$

Operation

ST(1) ← ST(1) * $\log_2$(ST(0) + 1.0);
PopRegisterStack;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	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.

HADDPD: Packed Double-FP Horizontal Add

Opcode	Instruction	Description
66,0F,7C,/r	HADDPD *xmm1*, *xmm2/m128*	Add horizontally packed DP FP numbers from *xmm2/m128* to *xmm1*.

Description

Adds the double-precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double-precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand.

See Figure 3-10.

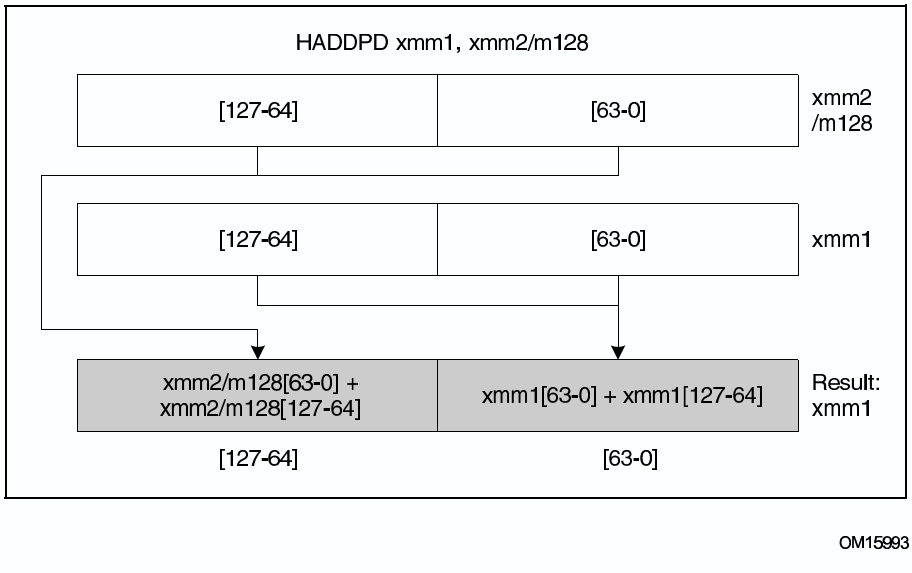

Figure 3-10. HADDPD: Packed Double-FP Horizontal Add

Operation

```
xmm1[63-0]    = xmm1[63-0]       + xmm1[127-64];
xmm1[127-64] = xmm2/m128[63-0] + xmm2/m128[127-64];
```

Intel C/C++ Compiler Intrinsic Equivalent

HADDPD __m128d _mm_hadd_pd(__m128d a, __m128d b)

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0);
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).

	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.
#PF(fault-code)	For a page fault.

HADDPS: Packed Single-FP Horizontal Add

Opcode	Instruction	Description
F2,0F,7C,/r	HADDPS *xmm1*, *xmm2/m128*	Add horizontally packed SP FP numbers from *xmm2/m128* to *xmm1*.

Description

Adds the single-precision floating-point values in the first and second dwords of the destination operand and stores the result in the first dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single-precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand.

See Figure 3-11.

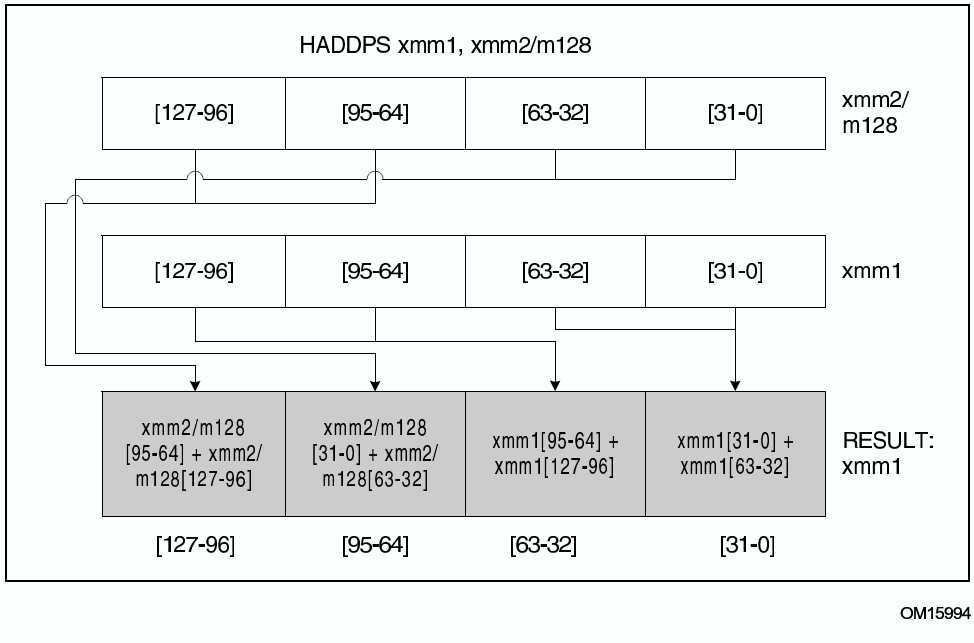

Figure 3-11. HADDPS: Packed Single-FP Horizontal Add

Operation

```
xmm1[31-0] = xmm1[31-0] + xmm1[63-32];
xmm1[63-32] = xmm1[95-64] + xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] + xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] + xmm2/m128[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
HADDPS __m128 _mm_hadd_ps(__m128 a, __m128 b)
```

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Virtual 8086 Mode Exceptions

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

#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.
#PF(fault-code)	For a page fault.

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.

When a HLT instruction is executed on an IA-32 processor supporting Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

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.

HSUBPD: Packed Double-FP Horizontal Subtract

Opcode	Instruction	Description
66,0F,7D,/r	HSUBPD *xmm1*, *xmm2/m128*	Subtract horizontally packed DP FP numbers in *xmm2/m128* from *xmm1.*

Description

The HSUBPD instruction subtracts horizontally the packed DP FP numbers of both operands.

Subtracts the double-precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double-precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand.

See Figure 3-12.

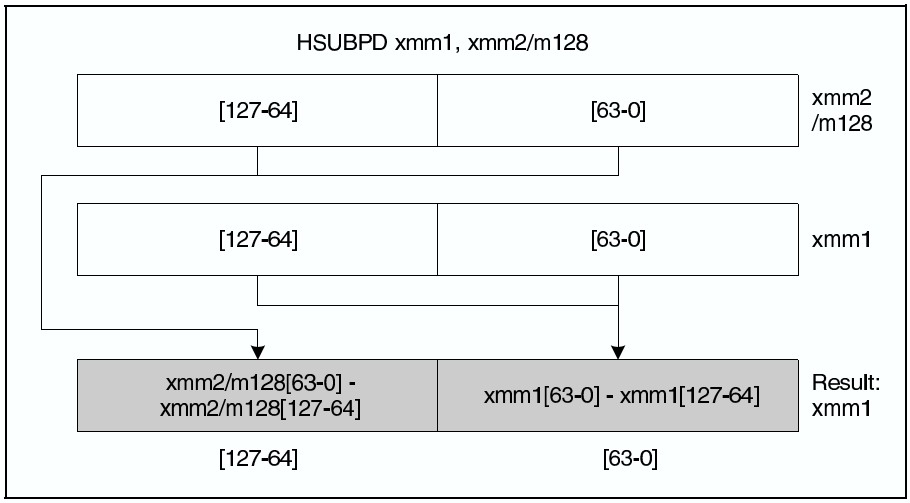

Figure 3-12. HSUBPD: Packed Double-FP Horizontal Subtract

Operation

```
xmm1[63-0]    = xmm1[63-0] - xmm1[127-64];
```

```
xmm1[127-64] = xmm2/m128[63-0] - xmm2/m128[127-64];
```

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPD __m128d _mm_hsub_pd(__m128d a, __m128d b)

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.

	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.
#PF(fault-code)	For a page fault.

HSUBPS: Packed Single-FP Horizontal Subtract

Opcode	Instruction	Description
F2,0F,7D,/r	HSUBPS *xmm1*, *xmm2/m128*	Subtract horizontally packed SP FP numbers in *xmm2/m128* from *xmm1*.

Description

Subtracts the single-precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single-precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

See Figure 3-13.

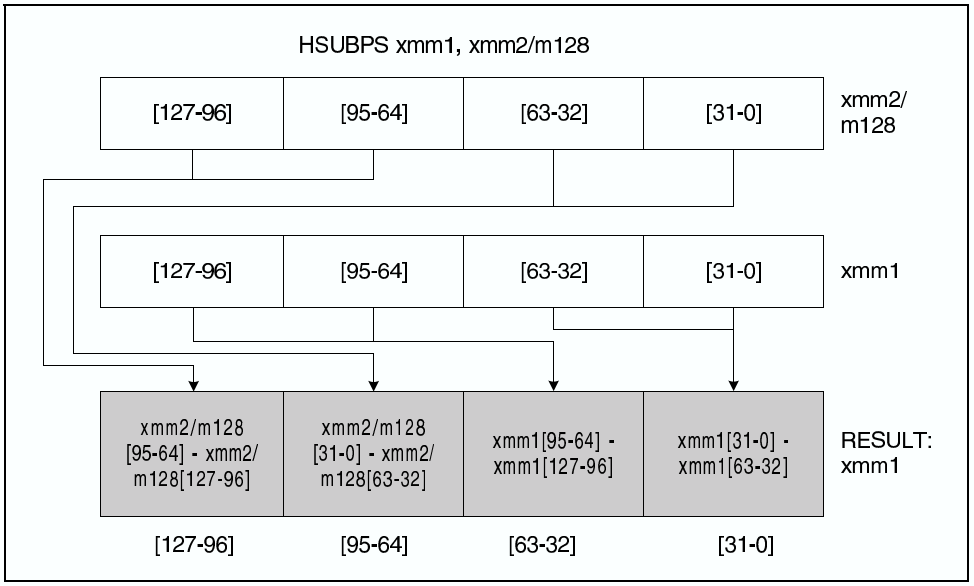

Figure 3-13. HSUBPS: Packed Single-FP Horizontal Subtract

Operation

```
xmm1[31-0] = xmm1[31-0] - xmm1[63-32];
xmm1[63-32] = xmm1[95-64] - xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
HSUBPS __m128 _mm_hsub_ps(__m128 a, __m128 b)
```

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Virtual 8086 Mode Exceptions

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

#XM	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 1).
#UD	If CR0.EM = 1.
	For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT = 0).
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.
#PF(fault-code)	For a page fault.

IDIV—Signed Divide

Opcode	Instruction	Description
F6 /7	IDIV *r/m8*	Signed divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
F7 /7	IDIV *r/m16*	Signed divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /7	IDIV *r/m32*	Signed divide EDX:EAX by *r/m32*, 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:

Table 3-51. IDIV Results

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	–128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{32} - 1$

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 OperandSize = 8 (* word/byte operation *)
    THEN
        temp ← AX / SRC; (* signed division *)
        IF (temp > 7FH) OR (temp < 80H)
        (* if a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* divide error *) ;
            ELSE
                AL ← temp;
                AH ← AX SignedModulus SRC;
        FI;
    ELSE
        IF OperandSize = 16 (* doubleword/word operation *)
```

```
            THEN
                temp ← DX:AX / SRC; (* signed division *)
                IF (temp > 7FFFH) OR (temp < 8000H)
                (* if a positive result is greater than 7FFFH *)
                (* or a negative result is less than 8000H *)
                    THEN #DE; (* divide error *) ;
                    ELSE
                        AX ← temp;
                        DX ← DX:AX SignedModulus SRC;
                FI;
            ELSE (* quadword/doubleword operation *)
                temp ← EDX:EAX / SRC; (* signed division *)
                IF (temp > 7FFFFFFFH) OR (temp < 80000000H)
                (* if a positive result is greater than 7FFFFFFFH *)
                (* or a negative result is less than 80000000H *)
                    THEN #DE; (* divide error *) ;
                    ELSE
                        EAX ← temp;
                        EDX ← EDXE:AX 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.

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 *r/m8*	AX← AL ∗ *r/m* byte.
F7 /5	IMUL *r/m16*	DX:AX ← AX ∗ *r/m* word.
F7 /5	IMUL *r/m32*	EDX:EAX ← EAX ∗ *r/m* doubleword.
0F AF /*r*	IMUL *r16,r/m16*	word register ← word register ∗ *r/m* word.
0F AF /*r*	IMUL *r32,r/m32*	doubleword register ← doubleword register ∗ *r/m* doubleword.
6B /*r ib*	IMUL *r16,r/m16,imm8*	word register ← *r/m16* ∗ sign-extended immediate byte.
6B /*r ib*	IMUL *r32,r/m32,imm8*	doubleword register ← *r/m32* ∗ sign-extended immediate byte.
6B /*r ib*	IMUL *r16,imm8*	word register ← word register ∗ sign-extended immediate byte.
6B /*r ib*	IMUL *r32,imm8*	doubleword register ← doubleword register ∗ sign-extended immediate byte.
69 /*r iw*	IMUL *r16,r/m16,imm16*	word register ← *r/m16* ∗ immediate word.
69 /*r id*	IMUL *r32,r/m32,imm32*	doubleword register ← *r/m32* ∗ immediate doubleword.
69 /*r iw*	IMUL *r16,imm16*	word register ← *r/m16* ∗ immediate word.
69 /*r id*	IMUL *r32,imm32*	doubleword register ← *r/m32* ∗ 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.

The CF and OF flags are set when significant bit (including the sign bit) are carried into the upper half of the result. The CF and OF flags are cleared when the result (including the sign bit) fits exactly in the lower half of the result.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

Operation

```
IF (NumberOfOperands = 1)
    THEN IF (OperandSize = 8)
        THEN
            AX ← AL * SRC  (* signed multiplication *)
            IF AL = AX
                THEN CF ← 0; OF ← 0;
                ELSE CF ← 1; OF ← 1;
            FI;
        ELSE IF OperandSize = 16
            THEN
                DX:AX ← AX * SRC  (* signed multiplication *)
                IF sign_extend_to_32 (AX) = DX:AX
                    THEN CF ← 0; OF ← 0;
                    ELSE CF ← 1; OF ← 1;
                FI;
            ELSE (* OperandSize = 32 *)
                EDX:EAX ← EAX * SRC  (* signed multiplication *)
                IF EAX = EDX:EAX
                    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 *)
            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 *ib*	IN AL,*imm8*	Input byte from *imm8* I/O port address into AL.
E5 *ib*	IN AX,*imm8*	Input byte from *imm8* I/O port address into AX.
E5 *ib*	IN EAX,*imm8*	Input byte from *imm8* 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 13, *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.

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 *r/m8*	Increment *r/m* byte by 1.
FF /0	INC *r/m16*	Increment *r/m* word by 1.
FF /0	INC *r/m32*	Increment *r/m* doubleword by 1.
40+ *rw*	INC *r16*	Increment word register by 1.
40+ *rd*	INC *r32*	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 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.

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.

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 13, *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;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

	If the destination is located in a non-writable segment.
	If an illegal memory operand effective address in the ES segments is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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 *ib*	INT *imm8*	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.

The interrupt vector number specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and

a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the "Operation" section for this instruction (except #GP).

Table 3-52. Decision Table

PE	0	1	1	1	1	1	1	1
VM	–	–	–	–	–	0	1	1
IOPL	–	–	–	–	–	–	<3	=3
DPL/CPL RELATIONSHIP	–	DPL< CPL	–	DPL> CPL	DPL= CPL or C	DPL< CPL & NC	–	–
INTERRUPT TYPE	–	S/W	–	–	–	–	–	–
GATE TYPE	–	–	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

NOTES:

– Don't Care.

Y Yes, Action Taken.

Blank Action Not Taken.

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 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operation

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts 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;
    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;
    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;
    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;
        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 *)
        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 set 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;
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;
NT ← 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;

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;
    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; (*interrupt flag set to 0: disabled*)
    FI;
    TF ← 0;
    NT ← 0;
    VM ← 0;
    RF ← 0;
END;
```

Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

Protected Mode Exceptions

#GP(0)	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 an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the interrupt vector number is outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT *n*, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
#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, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.
#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 pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is null.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.

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.
#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; it may be implemented differently on different families of 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.

Virtual-8086 Mode Exceptions

#GP(0)	The INVD instruction cannot be executed in virtual-8086 mode.

INVLPG—Invalidate TLB Entry

Opcode	Instruction	Description
0F 01/7	INVLPG m	Invalidate TLB Entry for page that contains *m.*

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 different families of 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 preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack). As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or inter-

rupted 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
                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;
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;
    CPL ← 3;
    (* 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 #TS (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 #TS (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 addrsses descriptor beyond descriptor table limit
        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
            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 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.

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 not busy.
	If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
	IF IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.

J*cc*—Jump if Condition Is Met

Opcode	Instruction	Description
77 *cb*	JA *rel8*	Jump short if above (CF=0 and ZF=0).
73 *cb*	JAE *rel8*	Jump short if above or equal (CF=0).
72 *cb*	JB *rel8*	Jump short if below (CF=1).
76 *cb*	JBE *rel8*	Jump short if below or equal (CF=1 or ZF=1).
72 *cb*	JC *rel8*	Jump short if carry (CF=1).
E3 *cb*	JCXZ *rel8*	Jump short if CX register is 0.
E3 *cb*	JECXZ *rel8*	Jump short if ECX register is 0.
74 *cb*	JE *rel8*	Jump short if equal (ZF=1).
7F *cb*	JG *rel8*	Jump short if greater (ZF=0 and SF=OF).
7D *cb*	JGE *rel8*	Jump short if greater or equal (SF=OF).
7C *cb*	JL *rel8*	Jump short if less (SF<>OF).
7E *cb*	JLE *rel8*	Jump short if less or equal (ZF=1 or SF<>OF).
76 *cb*	JNA *rel8*	Jump short if not above (CF=1 or ZF=1).
72 *cb*	JNAE *rel8*	Jump short if not above or equal (CF=1).
73 *cb*	JNB *rel8*	Jump short if not below (CF=0).
77 *cb*	JNBE *rel8*	Jump short if not below or equal (CF=0 and ZF=0).
73 *cb*	JNC *rel8*	Jump short if not carry (CF=0).
75 *cb*	JNE *rel8*	Jump short if not equal (ZF=0).
7E *cb*	JNG *rel8*	Jump short if not greater (ZF=1 or SF<>OF).
7C *cb*	JNGE *rel8*	Jump short if not greater or equal (SF<>OF).
7D *cb*	JNL *rel8*	Jump short if not less (SF=OF).
7F *cb*	JNLE *rel8*	Jump short if not less or equal (ZF=0 and SF=OF).
71 *cb*	JNO *rel8*	Jump short if not overflow (OF=0).
7B *cb*	JNP *rel8*	Jump short if not parity (PF=0).
79 *cb*	JNS *rel8*	Jump short if not sign (SF=0).
75 *cb*	JNZ *rel8*	Jump short if not zero (ZF=0).
70 *cb*	JO *rel8*	Jump short if overflow (OF=1).
7A *cb*	JP *rel8*	Jump short if parity (PF=1).
7A *cb*	JPE *rel8*	Jump short if parity even (PF=1).
7B *cb*	JPO *rel8*	Jump short if parity odd (PF=0).
78 *cb*	JS *rel8*	Jump short if sign (SF=1).
74 *cb*	JZ *rel8*	Jump short if zero (ZF = 1).
0F 87 *cw/cd*	JA *rel16/32*	Jump near if above (CF=0 and ZF=0).
0F 83 *cw/cd*	JAE *rel16/32*	Jump near if above or equal (CF=0).
0F 82 *cw/cd*	JB *rel16/32*	Jump near if below (CF=1).
0F 86 *cw/cd*	JBE *rel16/32*	Jump near if below or equal (CF=1 or ZF=1).
0F 82 *cw/cd*	JC *rel16/32*	Jump near if carry (CF=1).
0F 84 *cw/cd*	JE *rel16/32*	Jump near if equal (ZF=1).
0F 84 *cw/cd*	JZ *rel16/32*	Jump near if 0 (ZF=1).
0F 8F *cw/cd*	JG *rel16/32*	Jump near if greater (ZF=0 and SF=OF).

Opcode	Instruction	Description
0F 8D *cw/cd*	JGE *rel16/32*	Jump near if greater or equal (SF=OF).
0F 8C *cw/cd*	JL *rel16/32*	Jump near if less (SF<>OF).
0F 8E *cw/cd*	JLE *rel16/32*	Jump near if less or equal (ZF=1 or SF<>OF).
0F 86 *cw/cd*	JNA *rel16/32*	Jump near if not above (CF=1 or ZF=1).
0F 82 *cw/cd*	JNAE *rel16/32*	Jump near if not above or equal (CF=1).
0F 83 *cw/cd*	JNB *rel16/32*	Jump near if not below (CF=0).
0F 87 *cw/cd*	JNBE *rel16/32*	Jump near if not below or equal (CF=0 and ZF=0).
0F 83 *cw/cd*	JNC *rel16/32*	Jump near if not carry (CF=0).
0F 85 *cw/cd*	JNE *rel16/32*	Jump near if not equal (ZF=0).
0F 8E *cw/cd*	JNG *rel16/32*	Jump near if not greater (ZF=1 or SF<>OF).
0F 8C *cw/cd*	JNGE *rel16/32*	Jump near if not greater or equal (SF<>OF).
0F 8D *cw/cd*	JNL *rel16/32*	Jump near if not less (SF=OF).
0F 8F *cw/cd*	JNLE *rel16/32*	Jump near if not less or equal (ZF=0 and SF=OF).
0F 81 *cw/cd*	JNO *rel16/32*	Jump near if not overflow (OF=0).
0F 8B *cw/cd*	JNP *rel16/32*	Jump near if not parity (PF=0).
0F 89 *cw/cd*	JNS *rel16/32*	Jump near if not sign (SF=0).
0F 85 *cw/cd*	JNZ *rel16/32*	Jump near if not zero (ZF=0).
0F 80 *cw/cd*	JO *rel16/32*	Jump near if overflow (OF=1).
0F 8A *cw/cd*	JP *rel16/32*	Jump near if parity (PF=1).
0F 8A *cw/cd*	JPE *rel16/32*	Jump near if parity even (PF=1).
0F 8B *cw/cd*	JPO *rel16/32*	Jump near if parity odd (PF=0).
0F 88 *cw/cd*	JS *rel16/32*	Jump near if sign (SF=1).
0F 84 *cw/cd*	JZ *rel16/32*	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 J*cc* instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16,* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of –128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each J*cc* mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The J*cc* instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the J*cc* instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JECXZ and JCXZ instructions differ from the other J*cc* 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;
```

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 *cb*	JMP *rel8*	Jump short, relative, displacement relative to next instruction.
E9 *cw*	JMP *rel16*	Jump near, relative, displacement relative to next instruction.
E9 *cd*	JMP *rel32*	Jump near, relative, displacement relative to next instruction.
FF /4	JMP *r/m16*	Jump near, absolute indirect, address given in *r/m16.*
FF /4	JMP *r/m32*	Jump near, absolute indirect, address given in *r/m32.*
EA *cd*	JMP *ptr16:16*	Jump far, absolute, address given in operand.
EA *cp*	JMP *ptr16:32*	Jump far, absolute, address given in operand.
FF /5	JMP *m16:16*	Jump far, absolute indirect, address given in *m16:16.*
FF /5	JMP *m16:32*	Jump far, absolute indirect, address given in *m16:32.*

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, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (*rel8, rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target

operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 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 at 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 *)
    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 > 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;

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;
    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)	If offset in target operand, call gate, or TSS is beyond the code segment limits.
	If the segment selector in the destination operand, call gate, task gate, or TSS is null.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If the segment selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL
	(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.
	If the segment descriptor for selector in a call gate does not indicate it is a code segment.
	If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP (selector)	If the code segment being accessed is not present.
	If call gate, task gate, or TSS not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

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 *r16,r/m16*	*r16* ← *r/m16* masked by FF00H.
0F 02 /r	LAR *r32,r/m32*	*r32* ← *r/m32* 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 Table 3-53.
- 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.

Table 3-53. Segment and Gate Types

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 00FxFF00H;
                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 set to 0.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and 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.

LDDQU: Load Unaligned Integer 128 Bits

Opcode	Instruction	Description
F2,0F,F0,/r	LDDQU *xmm*, *mem*	Load data from *mem* and return 128 bits in an *xmm* register.

Description

The instruction is *functionally similar* to MOVDQU xmm, m128 for loading from memory. That is: 16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 16-byte boundary. Up to 32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by LDDQU be modified and stored to the same location, use MOVDQU or MOVDQA instead of LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the MOVDQA instruction.

Implementation Notes

- If the source is aligned to a 16-byte boundary, based on the implementation, the 16 bytes may be loaded more than once. For that reason, the usage of LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using MOVDQU.
- This instruction is a replacement for MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use MOVDQA store-load pairs when data is 128-bit aligned or MOVDQU store-load pairs when data is 128-bit unaligned.
- If the memory address is not aligned on 16-byte boundary, some implementations may load up to 32 bytes and return 16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.

Operation

```
xmm[127-0] = m128;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
HADDPS __m128i _mm_lddqu_si128(__m128i const *p)
```

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 bit in CR0 is set.
#UD	If CR4.OSFXSR(bit 9) = 0.
	If CR0.EM = 1.
	If CPUID.SSE3(ECX bit 0) = 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 any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE3(ECX bit 0) = 0.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

LDMXCSR—Load MXCSR Register

Opcode	Instruction	Description
0F AE /2	LDMXCSR *m32*	Load MXCSR register from *m32*.

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 an SIMD floating-point exception mask bit and sets the corresponding exception flag bit, an 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

```
MXCSR ← 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.
	For an attempt to set reserved bits in MXCSR.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If 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.

#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 would lie outside of the effective address space from 0 to FFFFH.
	For an attempt to set reserved bits in MXCSR.
#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

Opcode	Instruction	Description
C5 /r	LDS *r16,m16:16*	Load DS:*r16* with far pointer from memory.
C5 /r	LDS *r32,m16:32*	Load DS:*r32* with far pointer from memory.
0F B2 /r	LSS *r16,m16:16*	Load SS:*r16* with far pointer from memory.
0F B2 /r	LSS *r32,m16:32*	Load SS:*r32* with far pointer from memory.
C4 /r	LES *r16,m16:16*	Load ES:*r16* with far pointer from memory.
C4 /r	LES *r32,m16:32*	Load ES:*r32* with far pointer from memory.
0F B4 /r	LFS *r16,m16:16*	Load FS:*r16* with far pointer from memory.
0F B4 /r	LFS *r32,m16:32*	Load FS:*r32* with far pointer from memory.
0F B5 /r	LGS *r16,m16:16*	Load GS:*r16* with far pointer from memory.
0F B5 /r	LGS *r32,m16:32*	Load GS:*r32* 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 SegementSelector = 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
            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.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL.

	If the DS, ES, FS, or GS register is being loaded with a non-null segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or 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 *r16,m*	Store effective address for *m* in register *r16*.
8D /r	LEA *r32,m*	Store effective address for *m* in register *r32*.

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Table 3-54. Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

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
```

```
            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.
#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 Far Pointer

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

LFENCE—Load Fence

Opcode	Instruction	Description
0F 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 Far Pointer

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.