

IA-32 Intel® Architecture Software Developer's Manual

Volume 2: Instruction Set Reference

NOTE: The *IA-32 Intel Architecture Software Developer's Manual* consists of three volumes: *Basic Architecture*, Order Number 245470-012; *Instruction Set Reference*, Order Number 245471-012; and the *System Programming Guide*, Order Number 245472-012. Please refer to all three volumes when evaluating your design needs.

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

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

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

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.

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 - 2003 Intel Corporation

CONTENTS

	PAGE
CHAPTER 1	
ABOUT THIS MANUAL	
1.1.	IA-32 PROCESSORS COVERED IN THIS MANUAL 1-1
1.2.	OVERVIEW OF THE <i>IA-32 INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 2: INSTRUCTION SET REFERENCE</i> 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 1-4
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.	INSTRUCTION PREFIXES 2-1
2.3.	OPCODE 2-3
2.4.	MODR/M AND SIB BYTES 2-3
2.5.	DISPLACEMENT AND IMMEDIATE BYTES 2-4
2.6.	ADDRESSING-MODE ENCODING OF MODR/M AND SIB BYTES 2-4
CHAPTER 3	
INSTRUCTION SET REFERENCE	
3.1.	INTERPRETING THE INSTRUCTION REFERENCE PAGES 3-1
3.1.1.	Instruction Format 3-1
3.1.1.1.	Opcode Column 3-2
3.1.1.2.	Instruction Column 3-3
3.1.1.3.	Description Column 3-5
3.1.1.4.	Description 3-6
3.1.2.	Operation 3-6
3.1.3.	Intel® C/C++ Compiler Intrinsic Equivalents 3-9
3.1.3.1.	The Intrinsic API 3-10
3.1.3.2.	MMX™ Technology Intrinsic 3-10
3.1.3.3.	SSE and SSE2 Intrinsic 3-10
3.1.4.	Flags Affected 3-12
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
	AAS—ASCII Adjust AL After Subtraction 3-19

	PAGE
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
AND—Logical AND	3-32
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating- Point Values	3-34
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values	3-36
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values	3-38
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating- Point Values	3-40
ARPL—Adjust RPL Field of Segment Selector	3-42
BOUND—Check Array Index Against Bounds	3-44
BSF—Bit Scan Forward	3-46
BSR—Bit Scan Reverse	3-48
BSWAP—Byte Swap	3-50
BT—Bit Test	3-51
BTC—Bit Test and Complement	3-53
BTR—Bit Test and Reset	3-55
BTS—Bit Test and Set	3-57
CALL—Call Procedure	3-59
CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword	3-70
CDQ—Convert Double to Quad	3-71
CLC—Clear Carry Flag	3-72
CLD—Clear Direction Flag	3-73
CLFLUSH—Flush Cache Line	3-74
CLI — Clear Interrupt Flag	3-76
CLTS—Clear Task-Switched Flag in CR0	3-79
CMC—Complement Carry Flag	3-80
CMOVcc—Conditional Move	3-81
CMP—Compare Two Operands	3-85
CMPPD—Compare Packed Double-Precision Floating-Point Values	3-87
CMPPS—Compare Packed Single-Precision Floating-Point Values	3-92
CMPS/CMP SB/CMP SW/CMP SD—Compare String Operands	3-96
CMPSD—Compare Scalar Double-Precision Floating-Point Values	3-99
CMPSS—Compare Scalar Single-Precision Floating-Point Values	3-103
CMPXCHG—Compare and Exchange	3-107
CMPXCHG8B—Compare and Exchange 8 Bytes	3-109
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	3-111
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	3-114
CPUID—CPU Identification	3-117

	PAGE
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	3-136
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	3-138
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	3-140
CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	3-142
CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values	3-144
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	3-146
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	3-148
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	3-150
CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values	3-152
CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	3-154
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer	3-156
CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value	3-158
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value	3-160
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value	3-162
CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value	3-164
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer	3-166
CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	3-168
CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	3-170
CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	3-172
CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	3-174
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer	3-176
CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer	3-178
CWD/CDQ—Convert Word to Doubleword/Convert Doubleword to Quadword	3-180
CWDE—Convert Word to Doubleword	3-181

	PAGE
DAA—Decimal Adjust AL after Addition	3-182
DAS—Decimal Adjust AL after Subtraction	3-184
DEC—Decrement by 1	3-186
DIV—Unsigned Divide	3-188
DIVPD—Divide Packed Double-Precision Floating-Point Values	3-191
DIVPS—Divide Packed Single-Precision Floating-Point Values	3-193
DIVSD—Divide Scalar Double-Precision Floating-Point Values	3-195
DIVSS—Divide Scalar Single-Precision Floating-Point Values	3-197
EMMS—Empty MMX Technology State	3-199
ENTER—Make Stack Frame for Procedure Parameters	3-200
F2XM1—Compute $2x^{-1}$	3-203
FABS—Absolute Value	3-205
FADD/FADDP/FIADD—Add	3-206
FBLD—Load Binary Coded Decimal	3-209
FBSTP—Store BCD Integer and Pop	3-211
FCBS—Change Sign	3-214
FCLEX/FNCLEX—Clear Exceptions	3-215
FCMOVcc—Floating-Point Conditional Move	3-217
FCOM/FCOMP/FCOMPP—Compare Floating Point Values	3-219
FCOMI/FCOMIP/ FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS	3-222
FCOS—Cosine	3-225
FDECSTP—Decrement Stack-Top Pointer	3-227
FDIV/FDIVP/FIDIV—Divide	3-228
FDIVR/FDIVRP/FIDIVR—Reverse Divide	3-232
FFREE—Free Floating-Point Register	3-236
FICOM/FICOMP—Compare Integer	3-237
FILD—Load Integer	3-239
FINCSTP—Increment Stack-Top Pointer	3-241
FINIT/FNINIT—Initialize Floating-Point Unit	3-242
FIST/FISTP—Store Integer	3-244
FLD—Load Floating Point Value	3-247
FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant	3-249
FLDCW—Load x87 FPU Control Word	3-251
FLDENV—Load x87 FPU Environment	3-253
FMUL/FMULP/FIMUL—Multiply	3-255
FNOP—No Operation	3-258
FPATAN—Partial Arctangent	3-259
FPREM—Partial Remainder	3-261
FPREM1—Partial Remainder	3-264
FPTAN—Partial Tangent	3-267
FRNDINT—Round to Integer	3-269
FRSTOR—Restore x87 FPU State	3-270
FSAVE/FNSAVE—Store x87 FPU State	3-272
FSCALE—Scale	3-275
FSIN—Sine	3-277
FSINCOS—Sine and Cosine	3-279

	PAGE
FSQRT—Square Root	3-281
FST/FSTP—Store Floating Point Value	3-283
FSTCW/FNSTCW—Store x87 FPU Control Word	3-286
FSTENV/FNSTENV—Store x87 FPU Environment	3-288
FSTSW/FNSTSW—Store x87 FPU Status Word	3-291
FSUB/FSUBP/FISUB—Subtract	3-294
FSUBR/FSUBRP/FISUBR—Reverse Subtract	3-297
FTST—TEST	3-300
FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values	3-302
FWAIT—Wait	3-305
FXAM—Examine	3-306
FXCH—Exchange Register Contents	3-308
FXRSTOR—Restore x87 FPU, MMX Technology, SSE, and SSE2 State	3-310
FXSAVE—Save x87 FPU, MMX Technology, SSE, and SSE2 State	3-312
FTRACT—Extract Exponent and Significand	3-318
FYL2X—Compute $y * \log_2 x$	3-320
FYL2XP1—Compute $y * \log_2(x + 1)$	3-322
HLT—Halt	3-324
IDIV—Signed Divide	3-325
IMUL—Signed Multiply	3-328
IN—Input from Port	3-332
INC—Increment by 1	3-334
INS/INSB/INSW/INSD—Input from Port to String	3-336
INT n/INTO/INT 3—Call to Interrupt Procedure	3-339
INVD—Invalidate Internal Caches	3-351
INVLPG—Invalidate TLB Entry	3-353
IRET/IRETD—Interrupt Return	3-354
Jcc—Jump if Condition Is Met	3-362
JMP—Jump	3-366
LAHF—Load Status Flags into AH Register	3-373
LAR—Load Access Rights Byte	3-374
LDMXCSR—Load MXCSR Register	3-377
LDS/LES/LFS/LGS/LSS—Load Far Pointer	3-379
LEA—Load Effective Address	3-382
LEAVE—High Level Procedure Exit	3-384
LES—Load Full Pointer	3-386
LFENCE—Load Fence	3-387
LFS—Load Full Pointer	3-388
LGDT/LIDT—Load Global/Interrupt Descriptor Table Register	3-389
LGS—Load Full Pointer	3-391
LLDT—Load Local Descriptor Table Register	3-392
LIDT—Load Interrupt Descriptor Table Register	3-394
LMSW—Load Machine Status Word	3-395
LOCK—Assert LOCK# Signal Prefix	3-397
LODS/LODSB/LODSW/LODSD—Load String	3-399
LOOP/LOOPcc—Loop According to ECX Counter	3-402

	PAGE
LSL—Load Segment Limit	3-405
LSS—Load Full Pointer	3-408
LTR—Load Task Register	3-409
MASKMOVDQU—Store Selected Bytes of Double Quadword	3-411
MASKMOVQ—Store Selected Bytes of Quadword	3-413
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values	3-416
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values	3-419
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value	3-422
MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value . . .	3-425
MFENCE—Memory Fence	3-428
MINPD—Return Minimum Packed Double-Precision Floating-Point Values . .	3-429
MINPS—Return Minimum Packed Single-Precision Floating-Point Values . .	3-432
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value . . .	3-435
MINSS—Return Minimum Scalar Single-Precision Floating-Point Value	3-438
MOV—Move	3-441
MOV—Move to/from Control Registers	3-446
MOV—Move to/from Debug Registers	3-448
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values . .	3-450
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values . . .	3-452
MOVD—Move Doubleword	3-454
MOVDQA—Move Aligned Double Quadword	3-457
MOVDQU—Move Unaligned Double Quadword	3-459
MOVDQ2Q—Move Quadword from XMM to MMX Technology Register	3-461
MOVHLPs—Move Packed Single-Precision Floating-Point Values High to Low	3-462
MOVHPD—Move High Packed Double-Precision Floating-Point Value	3-463
MOVHPS—Move High Packed Single-Precision Floating-Point Values	3-465
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High	3-467
MOVLPD—Move Low Packed Double-Precision Floating-Point Value	3-468
MOVLPS—Move Low Packed Single-Precision Floating-Point Values	3-470
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask . .	3-472
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask . .	3-474
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint	3-476
MOVNTI—Store Doubleword Using Non-Temporal Hint	3-478
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint	3-480
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint	3-482
MOVNTQ—Store of Quadword Using Non-Temporal Hint	3-484
MOVQ—Move Quadword	3-486
MOVQ2DQ—Move Quadword from MMX Technology to XMM Register	3-488
MOVSB/MOVSb/MOVSW/MOVSD—Move Data from String to String	3-489
MOVSD—Move Scalar Double-Precision Floating-Point Value	3-492

	PAGE
MOVSS—Move Scalar Single--Precision Floating-Point Values	3-495
MOVSX—Move with Sign-Extension	3-498
MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values	3-499
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values	3-501
MOVZX—Move with Zero-Extend	3-503
MUL—Unsigned Multiply	3-504
MULPD—Multiply Packed Double-Precision Floating-Point Values	3-506
MULPS—Multiply Packed Single-Precision Floating-Point Values	3-508
MULSD—Multiply Scalar Double-Precision Floating-Point Values	3-510
MULSS—Multiply Scalar Single-Precision Floating-Point Values	3-512
NEG—Two's Complement Negation	3-514
NOP—No Operation	3-516
NOT—One's Complement Negation	3-517
OR—Logical Inclusive OR	3-519
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	3-521
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values	3-523
OUT—Output to Port	3-525
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port	3-527
PACKSSWB/PACKSSDW—Pack with Signed Saturation	3-530
PACKUSWB—Pack with Unsigned Saturation	3-534
PADDB/PADDW/PADD—Add Packed Integers	3-537
PADDQ—Add Packed Quadword Integers	3-541
PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation	3-543
PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation	3-546
PAND—Logical AND	3-549
PANDN—Logical AND NOT	3-551
PAUSE—Spin Loop Hint	3-553
PAVGB/PAVGW—Average Packed Integers	3-554
PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal	3-557
PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than	3-561
PEXTRW—Extract Word	3-565
PINSRW—Insert Word	3-568
PMADDWD—Multiply and Add Packed Integers	3-571
PMAWSW—Maximum of Packed Signed Word Integers	3-574
PMAWSB—Maximum of Packed Unsigned Byte Integers	3-577
PMINSW—Minimum of Packed Signed Word Integers	3-580
PMINSB—Minimum of Packed Unsigned Byte Integers	3-583
PMOVBMSKB—Move Byte Mask	3-586
PMULHUW—Multiply Packed Unsigned Integers and Store High Result	3-588
PMULHW—Multiply Packed Signed Integers and Store High Result	3-591
PMULLW—Multiply Packed Signed Integers and Store Low Result	3-594
PMULUDQ—Multiply Packed Unsigned Doubleword Integers	3-597

	PAGE
POP—Pop a Value from the Stack	3-599
POPA/POPAD—Pop All General-Purpose Registers	3-604
POPF/POPFD—Pop Stack into EFLAGS Register	3-606
POR—Bitwise Logical OR	3-609
PREFETCHh—Prefetch Data Into Caches	3-611
PSADBW—Compute Sum of Absolute Differences	3-613
PSHUFD—Shuffle Packed Doublewords	3-616
PSHUFHW—Shuffle Packed High Words	3-618
PSHUFLW—Shuffle Packed Low Words	3-620
PSHUFW—Shuffle Packed Words	3-622
PSLLDQ—Shift Double Quadword Left Logical	3-624
PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical	3-625
PSRAW/PSRAD—Shift Packed Data Right Arithmetic	3-630
PSRLDQ—Shift Double Quadword Right Logical	3-634
PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical	3-635
PSUBB/PSUBW/PSUBD—Subtract Packed Integers	3-640
PSUBQ—Subtract Packed Quadword Integers	3-644
PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation	3-647
PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation	3-650
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ—Unpack High Data	3-653
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data	3-658
PUSH—Push Word or Doubleword Onto the Stack	3-663
PUSHA/PUSHAD—Push All General-Purpose Registers	3-666
PUSHF/PUSHFD—Push EFLAGS Register onto the Stack	3-668
PXOR—Logical Exclusive OR	3-670
RCL/RCR/ROL/ROR—Rotate	3-672
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values	3-677
RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values	3-679
RDMSR—Read from Model Specific Register	3-681
RDPMC—Read Performance-Monitoring Counters	3-682
RDTSC—Read Time-Stamp Counter	3-685
RDTSC—Read Time-Stamp Counter (Continued)	3-686
REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix	3-687
RET—Return from Procedure	3-690
ROL/ROR—Rotate	3-696
RSM—Resume from System Management Mode	3-697
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values	3-698
RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value	3-700
SAHF—Store AH into Flags	3-702

	PAGE
SAL/SAR/SHL/SHR—Shift	3-703
SBB—Integer Subtraction with Borrow	3-708
SCAS/SCASB/SCASW/SCASD—Scan String	3-710
SETcc—Set Byte on Condition	3-713
SFENCE—Store Fence	3-716
SGDT/SIDT—Store Global/Interrupt Descriptor Table Register	3-717
SHL/SHR—Shift Instructions	3-720
SHLD—Double Precision Shift Left	3-721
SHRD—Double Precision Shift Right	3-723
SHUFPS—Shuffle Packed Double-Precision Floating-Point Values	3-725
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values	3-728
SIDT—Store Interrupt Descriptor Table Register	3-731
SLDT—Store Local Descriptor Table Register	3-732
SMSW—Store Machine Status Word	3-734
SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values	3-736
SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values	3-738
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value	3-740
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value	3-742
STC—Set Carry Flag	3-744
STD—Set Direction Flag	3-745
STI—Set Interrupt Flag	3-746
STMXCSR—Store MXCSR Register State	3-750
STOS/STOSB/STOSW/STOSD—Store String	3-752
STR—Store Task Register	3-755
SUB—Subtract	3-756
SUBPD—Subtract Packed Double-Precision Floating-Point Values	3-758
SUBPS—Subtract Packed Single-Precision Floating-Point Values	3-760
SUBSD—Subtract Scalar Double-Precision Floating-Point Values	3-762
SUBSS—Subtract Scalar Single-Precision Floating-Point Values	3-764
SYSENTER—Fast System Call	3-766
SYSEXIT—Fast Return from Fast System Call	3-770
TEST—Logical Compare	3-773
UCOMISD—Unordered Compare Scalar Double-Precision Floating- Point Values and Set EFLAGS	3-775
UCOMISS—Unordered Compare Scalar Single-Precision Floating- Point Values and Set EFLAGS	3-778
UD2—Undefined Instruction	3-781
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values	3-782
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values	3-785
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values	3-788

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values	3-791
VERR, VERW—Verify a Segment for Reading or Writing	3-794
WAIT/FWAIT—Wait	3-796
WBINVD—Write Back and Invalidate Cache	3-797
WRMSR—Write to Model Specific Register	3-799
XADD—Exchange and Add	3-801
XCHG—Exchange Register/Memory with Register	3-803
XLAT/XLATB—Table Look-up Translation	3-805
XOR—Logical Exclusive OR	3-807
XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values	3-809
XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values	3-811

APPENDIX A

OPCODE MAP

A.1. KEY TO ABBREVIATIONS	A-1
A.1.1. Codes for Addressing Method	A-1
A.1.2. Codes for Operand Type	A-3
A.1.3. Register Codes	A-3
A.2. OPCODE LOOK-UP EXAMPLES	A-3
A.2.1. One-Byte Opcode Instructions	A-4
A.2.2. Two-Byte Opcode Instructions	A-4
A.2.3. Opcode Map Notes	A-5
A.2.4. Opcode Extensions For One- And Two-byte Opcodes	A-12
A.2.5. Escape Opcode Instructions	A-14
A.2.5.1. Opcodes with ModR/M Bytes in the 00H through BFH Range	A-14
A.2.5.2. Opcodes with ModR/M Bytes outside the 00H through BFH Range	A-14
A.2.5.3. Escape Opcodes with D8 as First Byte	A-14
A.2.5.4. Escape Opcodes with D9 as First Byte	A-16
A.2.5.5. Escape Opcodes with DA as First Byte	A-17
A.2.5.6. Escape Opcodes with DB as First Byte	A-18
A.2.5.7. Escape Opcodes with DC as First Byte	A-20
A.2.5.8. Escape Opcodes with DD as First Byte	A-21
A.2.5.9. Escape Opcodes with DE as First Byte	A-23
A.2.5.10. Escape Opcodes with DF As First Byte	A-24

APPENDIX B

INSTRUCTION FORMATS AND ENCODINGS

B.1. MACHINE INSTRUCTION FORMAT	B-1
B.1.1. Reg Field (reg)	B-2
B.1.2. Encoding of Operand Size Bit (w)	B-3
B.1.3. Sign Extend (s) Bit	B-3
B.1.4. Segment Register Field (sreg)	B-4
B.1.5. Special-Purpose Register (eee) Field	B-4
B.1.6. Condition Test Field (ttn)	B-5
B.1.7. Direction (d) Bit	B-5
B.2. GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS	B-6
B.3. 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

	PAGE
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.8. FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS	B-46

APPENDIX C

INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

C.1. SIMPLE INTRINSICS	C-3
C.2. COMPOSITE INTRINSICS	C-31

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].	3-9
Figure 3-2. Memory Bit Indexing	3-9
Figure 3-3. Version Information in the EAX Register	3-120
Figure 3-4. Extended Feature Flags Returned in ECX Register	3-122
Figure 3-5. Feature Information in the EDX Register	3-123
Figure 3-6. Operation of the PACKSSDW Instruction Using 64-bit Operands.	3-530
Figure 3-7. PMADDWD Execution Model Using 64-bit Operands	3-571
Figure 3-8. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands	3-588
Figure 3-9. PMULLU Instruction Operation Using 64-bit Operands	3-594
Figure 3-10. PSADBW Instruction Operation Using 64-bit Operands.	3-613
Figure 3-11. PSHUFD Instruction Operation.	3-616
Figure 3-12. PSSLW, PSLLD, and PSSLQ Instruction Operation Using 64-bit Operand	3-625
Figure 3-13. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand	3-630
Figure 3-14. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand	3-635
Figure 3-15. PUNPCKHBW Instruction Operation Using 64-bit Operands.	3-653
Figure 3-16. PUNPCKLBW Instruction Operation Using 64-bit Operands	3-658
Figure 3-17. SHUFPS Shuffle Operation	3-725
Figure 3-18. SHUFPD Shuffle Operation	3-728
Figure 3-19. UNPCKHPD Instruction High Unpack and Interleave Operation	3-782
Figure 3-20. UNPCKHPS Instruction High Unpack and Interleave Operation	3-785
Figure 3-21. UNPCKLPD Instruction Low Unpack and Interleave Operation	3-788
Figure 3-22. UNPCKLPS Instruction Low Unpack and Interleave Operation	3-791
Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3).	A-12
Figure B-1. General Machine Instruction Format	B-1

	PAGE
Table 2-1.	16-Bit Addressing Forms with the ModR/M Byte 2-5
Table 2-2.	32-Bit Addressing Forms with the ModR/M Byte 2-6
Table 2-3.	32-Bit Addressing Forms with the SIB Byte 2-7
Table 3-1.	Register Encodings Associated with the +rb, +rw, and +rd Nomenclature . . . 3-3
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 3-76
Table 3-6.	Comparison Predicate for CMPPD and CMPPS Instructions 3-87
Table 3-7.	Information Returned by CPUID Instruction 3-118
Table 3-8.	Highest CPUID Source Operand for IA-32 Processors 3-119
Table 3-9.	Processor Type Field 3-120
Table 3-10.	Extended Feature Flags Returned in ECX Register 3-122
Table 3-11.	CPUID Feature Flags Returned in EDX Register 3-124
Table 3-12.	Encoding of Cache and TLB Descriptors 3-127
Table 3-13.	Mapping of Brand Indices and IA-32 Processor Brand Strings 3-131
Table 3-14.	Processor Brand String Returned with First Pentium 4 Processor 3-133
Table 3-15.	Layout of FXSAVE and FXRSTOR Memory Region 3-312
Table 3-16.	Decision Table for STI Results 3-747
Table 3-17.	MSRs Used By the SYSENTER and SYSEXIT Instructions 3-766
Table A-1.	Notes on Instruction Set Encoding Tables A-5
Table A-2.	One-byte Opcode Map: 00H — F7H† A-6
Table A-3.	Two-byte Opcode Map: 00H — 77H (First Byte is 0FH)† A-8
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-14
Table A-6.	D8 Opcode Map When ModR/M Byte is Outside 00H to BFH1 A-15
Table A-7.	D9 Opcode Map When ModR/M Byte is Within 00H to BFH1 A-16
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-17
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-21
Table A-15.	DD Opcode Map When ModR/M Byte is Within 00H to BFH1 A-22
Table A-16.	DD Opcode Map When ModR/M Byte is Outside 00H to BFH1 A-22
Table A-17.	DE Opcode Map When ModR/M Byte is Within 00H to BFH1 A-23
Table A-18.	DE Opcode Map When ModR/M Byte is Outside 00H to BFH1 A-24
Table A-19.	DF Opcode Map When ModR/M Byte is Within 00H to BFH1 A-25
Table A-20.	DF Opcode Map When ModR/M Byte is Outside 00H to BFH1 A-25
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.	General Purpose Instruction Formats and Encodings B-6
Table B-11.	Pentium Family Instruction Formats and Encodings B-19
Table B-12.	Encoding of Granularity of Data Field (gg) B-20

	PAGE
Table B-13. MMX Instruction Formats and Encodings	B-20
Table B-14. Formats and Encodings of P6 Family Instructions	B-24
Table B-15. Formats and Encodings of SSE SIMD Floating-Point Instructions	B-25
Table B-16. Formats and Encodings of SSE SIMD Integer Instructions	B-31
Table B-17. Format and Encoding of the SSE Cacheability and Memory Ordering Instructions	B-32
Table B-18. Encoding of Granularity of Data Field (gg)	B-33
Table B-19. Formats and Encodings of the SSE2 SIMD Floating-Point Instructions	B-33
Table B-20. Formats and Encodings of the SSE2 SIMD Integer Instructions	B-40
Table B-21. Format and Encoding of the SSE2 Cacheability Instructions	B-45
Table B-22. General Floating-Point Instruction Formats	B-46
Table B-23. Floating-Point Instruction Formats and Encodings	B-47
Table C-1. Simple Intrinsics	C-3
Table C-2. Composite Intrinsics	C-31

1

About This Manual

CHAPTER 1

ABOUT THIS MANUAL

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

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

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

1.1. IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual includes information pertaining primarily to the most recent IA-32 processors, which include the Pentium® 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 micro-architecture, which include the Pentium Pro, Pentium II, and Pentium III processors. The Pentium 4 and Intel Xeon processors are based on the Intel® NetBurst™ micro-architecture.

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

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

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

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

Chapter 3 — Instruction Set Reference. Describes each of the IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. The general-purpose, x87 FPU, Intel MMX™ technology, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), and system instructions are included in this chapter.

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

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

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

1.3. NOTATIONAL CONVENTIONS

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

1.3.1. Bit and Byte Order

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

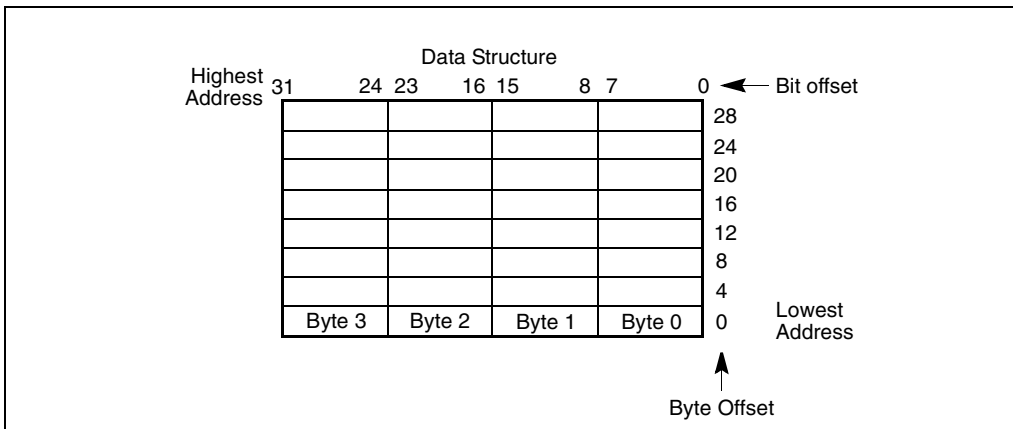


Figure 1-1. Bit and Byte Order

1.3.2. Reserved Bits and Software Compatibility

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

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

NOTE

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

1.3.3. Instruction Operands

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

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

where:

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

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

For example:

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

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

1.3.4. Hexadecimal and Binary Numbers

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

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

1.3.5. Segmented Addressing

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

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

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed to 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 break-points, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below.

#PF(fault code)

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

#GP(0)

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

1.4. RELATED LITERATURE

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

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

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

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

2

Instruction Format

- 36H—SS segment override prefix (use with any branch instruction is reserved).
- 3EH—DS segment override prefix (use with any branch instruction is reserved).
- 26H—ES segment override prefix (use with any branch instruction is reserved).
- 64H—FS segment override prefix (use with any branch instruction is reserved).
- 65H—GS segment override prefix (use with any branch instruction is reserved).
- Branch hints:
 - 2EH—Branch not taken (used only with *Jcc* instructions).
 - 3EH—Branch taken (used only with *Jcc* instructions).
- Group 3
 - 66H—Operand-size override prefix.
- Group 4
 - 67H—Address-size override prefix.

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

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

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

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

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

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

NOTE

Some of the SSE and SSE2 instructions have three-byte opcodes. For these three-byte opcodes, the third opcode byte may be F2H, F3H, or 66H. For example, the SSE2 instruction CVTQ2PD has the three-byte opcode F3 0F E6. The third opcode byte of these three-byte opcodes should not be thought of as a prefix, even though it has the same encoding as the operand size prefix (66H) or one of the repeat prefixes (F2H and F3H). As described above, using the operand size and repeat prefixes with SSE and SSE2 instructions is reserved. It should also be noted that execution of SSE2 instructions on an Intel processor that does not support SSE2 (CPUID Feature flag register EDX bit 26 is clear) will result in unpredictable code execution.

2.3. OPCODE

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

2.4. MODR/M AND SIB BYTES

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

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

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

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

See Section 2.6., “Addressing-Mode Encoding of ModR/M and SIB Bytes”, for the encodings of the ModR/M and SIB bytes.

2.5. DISPLACEMENT AND IMMEDIATE BYTES

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

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

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

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

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

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

Across the top of Tables 2-1 and 2-2, the eight possible values of the 3-bit Reg/Opcode field are listed, in decimal (sixth row from top) and in binary (seventh row from top). The seventh row is labeled “REG=”, which represents the use of these 3 bits to give the location of a second operand, which must be a general-purpose, MMX 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 Tables 2-1 and 2-2 (under the label “Value of ModR/M Byte (in Hexadecimal)”) contains a 32 by 8 array giving all of the 256 values of the ModR/M byte, in hexadecimal. Bits 3, 4 and 5 are specified by the column of the table in which a byte resides, and the row specifies bits 0, 1 and 2, and also bits 6 and 7.

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

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

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows 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 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 8 general-purpose registers will be used as base is indicated across the top of the table, along with the corresponding values of the base field (bits 0, 1 and 2) in decimal and binary. The rows indicate which register is used as the index (determined by bits 3, 4 and 5) along with the scaling factor (determined by bits 6 and 7).

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

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

NOTE:

- 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:

<u>MOD bits</u>	<u>Effective Address</u>
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]



3

Instruction Set Reference

CHAPTER 3

INSTRUCTION SET REFERENCE

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

3.1. INTERPRETING THE INSTRUCTION REFERENCE PAGES

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

3.1.1. Instruction Format

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

CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement carry flag

3.1.1.1. OPCODE COLUMN

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

- **/digit**—A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the *r/m* (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r**—Indicates that the ModR/M byte of the instruction contains both a register operand and an *r/m* operand.
- **cb, cw, cd, cp**—A 1-byte (cb), 2-byte (cw), 4-byte (cd), or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id**—A 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.
- **+rb, +rw, +rd**—A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The register codes are given in Table 3-3.
- **+i**—A number used in floating-point instructions when one of the operands is ST(*i*) from the FPU register stack. The number *i* (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 3-1. Register Encodings Associated with the +rb, +rw, and +rd Nomenclature

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

3.1.1.2. INSTRUCTION COLUMN

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

- **rel8**—A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16 and rel32**—A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.
- **ptr16:16 and ptr16:32**—A far pointer, typically in a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction’s operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8**—One of the byte general-purpose registers AL, CL, DL, BL, AH, CH, DH, or BH.
- **r16**—One of the word general-purpose registers AX, CX, DX, BX, SP, BP, SI, or DI.
- **r32**—One of the doubleword general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.
- **imm8**—An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16**—An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.

- **imm32**—An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and -2,147,483,648 inclusive.
- **r/m8**—A byte operand that is either the contents of a byte general-purpose register (AL, BL, CL, DL, AH, BH, CH, and DH), or a byte from memory.
- **r/m16**—A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, BX, CX, DX, SP, BP, SI, and DI. The contents of memory are found at the address provided by the effective address computation.
- **r/m32**—A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. The contents of memory are found at the address provided by the effective address computation.
- **m**—A 16- or 32-bit operand in memory.
- **m8**—A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions and the XLAT instruction.
- **m16**—A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32**—A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m64**—A memory quadword operand in memory. This nomenclature is used only with the CMPXCHG8B instruction.
- **m128**—A memory double quadword operand in memory. This nomenclature is used only with the SSE and SSE2 instructions.
- **m16:16, m16:32**—A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32**—A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers.
- **moffs8, moffs16, moffs32**—A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the

instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

- **Sreg**—A segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.
- **m32fp, m64fp, m80fp**—A single-precision, double-precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.
- **m16int, m32int, m64int**—A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.
- **ST or ST(0)**—The top element of the FPU register stack.
- **ST(i)**—The i^{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.
- $A \leftarrow B$; indicates that the value of B is assigned to A.
- The symbols =, \neq , \geq , and \leq are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as $A = B$ 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  $\leftarrow$  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 this chapter for more information on the push operation.
- **Pop()** removes the value from the top of the stack and returns it. The statement $EAX \leftarrow Pop()$; assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute. See the “Operation” section in Chapter 3, “POP—Pop a Value from the Stack” for more information on the pop operation.
- **PopRegisterStack**—Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks**—Performs a task switch.
- **Bit(BitBase, BitOffset)**—Returns the value of a bit within a bit string, which is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register. An example, the function `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.

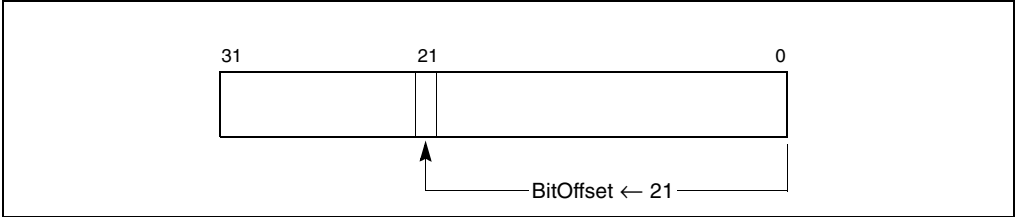


Figure 3-1. Bit Offset for BIT[EAX,21]

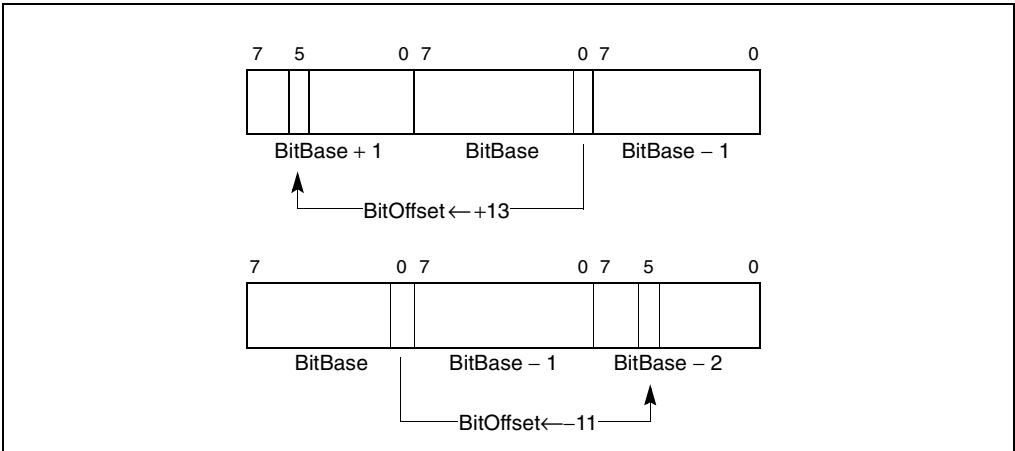


Figure 3-2. Memory Bit Indexing

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

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

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

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

3.1.3.1. THE INTRINSICS API

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

3.1.3.2. MMX™ TECHNOLOGY INTRINSICS

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

- Use `__m64` data only on the left-hand side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions (“+”, “>>”, and so on).
- Use `__m64` objects in aggregates, such as unions to access the byte elements and structures; the address of an `__m64` object may be taken.
- Use `__m64` data only with the MMX technology intrinsics described in this guide and the *Intel C/C++ Compiler User's Guide With Support for the Streaming SIMD Extensions 2* (Order Number 718195-2001). Refer to *Appendix C, Intel C/C++ Compiler Intrinsics and Functional Equivalents* for more information on using intrinsics.

3.1.3.3. SSE AND SSE2 INTRINSICS

The SSE and SSE2 intrinsics all make use of the XMM registers of the Pentium III, 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, and MMX technology when writing programs with the intrinsics.

Keep the following three important issues in mind:

- Certain intrinsics, such as `_mm_loadr_ps` and `_mm_cmpgt_ss`, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful of their implementation cost.
- Data loaded or stored as `__m128` objects must generally be 16-byte-aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), due to the nature of the instruction.
- The result of arithmetic operations acting on two NaN (Not a Number) arguments is undefined. Therefore, floating-point operations using NaN arguments 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 and SSE2 floating-point instructions.	Yes	Yes	Yes

NOTE:

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

3.1.7. Real-Address Mode Exceptions

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

3.1.8. Virtual-8086 Mode Exceptions

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

3.1.9. Floating-Point Exceptions

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

Table 3-3. x87 FPU Floating-Point Exceptions

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

3.1.10. SIMD Floating-Point Exceptions

The “SIMD Floating-Point Exceptions” section lists exceptions that can occur when an SSE and SSE2 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XF, vector number 19) being generated. Table 3-4 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to “SSE and SSE2 Exceptions”, in Chapter 11 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*.

Table 3-4. SIMD Floating-Point Exceptions

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 each of the IA-32 instructions.

AAA—ASCII Adjust After Addition

Opcode	Instruction	Description
37	AAA	ASCII adjust AL after addition

Description

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

If the addition produces a decimal carry, the AH register 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 $((AL \text{ AND } 0FH) > 9) \text{ OR } (AF = 1)$

THEN

AL \leftarrow AL + 6;

AH \leftarrow AH + 1;

AF \leftarrow 1;

CF \leftarrow 1;

ELSE

AF \leftarrow 0;

CF \leftarrow 0;

FI;

AL \leftarrow 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 <i>ib</i>	(No mnemonic)	Adjust AX before division to number base <i>imm8</i>

Description

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

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

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

Operation

tempAL \leftarrow AL;

tempAH \leftarrow AH;

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

AH \leftarrow 0

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

Flags Affected

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

Exceptions (All Operating Modes)

None.

AAM—ASCII Adjust AX After Multiply

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

Description

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

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

Operation

tempAL ← AL;

AH ← tempAL / *imm8*; (* *imm8* is set to 0AH for the 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 <i>ib</i>	ADC AL, <i>imm8</i>	Add with carry <i>imm8</i> to AL
15 <i>iw</i>	ADC AX, <i>imm16</i>	Add with carry <i>imm16</i> to AX
15 <i>id</i>	ADC EAX, <i>imm32</i>	Add with carry <i>imm32</i> to EAX
80 /2 <i>ib</i>	ADC <i>r/m8</i> , <i>imm8</i>	Add with carry <i>imm8</i> to <i>r/m8</i>
81 /2 <i>iw</i>	ADC <i>r/m16</i> , <i>imm16</i>	Add with carry <i>imm16</i> to <i>r/m16</i>
81 /2 <i>id</i>	ADC <i>r/m32</i> , <i>imm32</i>	Add with CF <i>imm32</i> to <i>r/m32</i>
83 /2 <i>ib</i>	ADC <i>r/m16</i> , <i>imm8</i>	Add with CF sign-extended <i>imm8</i> to <i>r/m16</i>
83 /2 <i>ib</i>	ADC <i>r/m32</i> , <i>imm8</i>	Add with CF sign-extended <i>imm8</i> into <i>r/m32</i>
10 / <i>r</i>	ADC <i>r/m8</i> , <i>r8</i>	Add with carry byte register to <i>r/m8</i>
11 / <i>r</i>	ADC <i>r/m16</i> , <i>r16</i>	Add with carry <i>r16</i> to <i>r/m16</i>
11 / <i>r</i>	ADC <i>r/m32</i> , <i>r32</i>	Add with CF <i>r32</i> to <i>r/m32</i>
12 / <i>r</i>	ADC <i>r8</i> , <i>r/m8</i>	Add with carry <i>r/m8</i> to byte register
13 / <i>r</i>	ADC <i>r16</i> , <i>r/m16</i>	Add with carry <i>r/m16</i> to <i>r16</i>
13 / <i>r</i>	ADC <i>r32</i> , <i>r/m32</i>	Add with CF <i>r/m32</i> to <i>r32</i>

Description

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

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

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

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

Operation

DEST ← DEST + SRC + CF;

Flags Affected

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

ADC—Add with Carry (Continued)

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

ADD—Add

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

Description

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

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

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

Operation

DEST ← DEST + SRC;

Flags Affected

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

ADD—Add (Continued)

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

ADDPD—Add Packed Double-Precision Floating-Point Values

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

Description

Performs a SIMD add of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD double-precision floating-point operation.

Operation

$$\text{DEST}[63-0] \leftarrow \text{DEST}[63-0] + \text{SRC}[63-0];$$

$$\text{DEST}[127-64] \leftarrow \text{DEST}[127-64] + \text{SRC}[127-64];$$

Intel C/C++ Compiler Intrinsic Equivalent

ADDPD `__m128d _mm_add_pd (m128d a, m128d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

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

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

ADDPS—Add Packed Single-Precision Floating-Point Values

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

Description

Performs a SIMD add of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD single-precision floating-point operation.

Operation

```
DEST[31-0] ← DEST[31-0] + SRC[31-0];
DEST[63-32] ← DEST[63-32] + SRC[63-32];
DEST[95-64] ← DEST[95-64] + SRC[95-64];
DEST[127-96] ← DEST[127-96] + SRC[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

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

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

ADDSD—Add Scalar Double-Precision Floating-Point Values

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

Description

Adds the low double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

Operation

DEST[63-0] ← DEST[63-0] + SRC[63-0];

* DEST[127-64] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalent

ADDSD __m128d _mm_add_sd (m128d a, m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

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

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

ADDSS—Add Scalar Single-Precision Floating-Point Values

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

Description

Adds the low single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

Operation

DEST[31-0] ← DEST[31-0] + SRC[31-0];

* DEST[127-32] remain unchanged *;

Intel C/C++ Compiler Intrinsic Equivalent

ADDSS __m128 _mm_add_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

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

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

AND—Logical AND

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

Description

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

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

Operation

DEST ← DEST AND SRC;

Flags Affected

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

AND—Logical AND (Continued)

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 <i>xmm1</i> , <i>xmm2/m128</i>	Bitwise logical AND of <i>xmm2/m128</i> and <i>xmm1</i> .

Description

Performs a bitwise logical AND of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

ANDPD __m128d _mm_and_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

Performs a bitwise logical AND of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

ANDPS __m128 _mm_and_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

Inverts the bits of the two packed double-precision floating-point values in the destination operand (first operand), performs a bitwise logical AND of the two packed double-precision floating-point values in the source operand (second operand) and the temporary inverted result, and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

ANDNPD __m128d _mm_andnot_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

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

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

If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

Inverts the bits of the four packed single-precision floating-point values in the destination operand (first operand), performs a bitwise logical AND of the four packed single-precision floating-point values in the source operand (second operand) and the temporary inverted result, and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

ANDNPS __m128 _mm_andnot_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

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

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

If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

ARPL—Adjust RPL Field of Segment Selector

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

Description

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then insures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program. (The segment selector for the application program's code segment can be read from the stack following a procedure call.)

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

Operation

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

Flags Affected

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

ARPL—Adjust RPL Field of Segment Selector (Continued)

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#UD	The ARPL instruction is not recognized in real-address mode.
-----	--

Virtual-8086 Mode Exceptions

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

BOUND—Check Array Index Against Bounds

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

Description

Determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. (When a this exception is generated, the saved return instruction pointer points to the BOUND instruction.)

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

Operation

```
IF (ArrayIndex < LowerBound OR ArrayIndex > 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.

BOUND—Check Array Index Against Bounds (Continued)

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

BSF—Bit Scan Forward

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

Description

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

Operation

```

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

```

Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

Protected Mode Exceptions

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

BSF—Bit Scan Forward (Continued)

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

BSR—Bit Scan Reverse

Opcode	Instruction	Description
0F BD	BSR <i>r16,r/m16</i>	Bit scan reverse on <i>r/m16</i>
0F BD	BSR <i>r32,r/m32</i>	Bit scan reverse on <i>r/m32</i>

Description

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

Operation

```

IF SRC = 0
  THEN
    ZF ← 1;
    DEST is undefined;
  ELSE
    ZF ← 0;
    temp ← OperandSize – 1;
    WHILE Bit(SRC, temp) = 0
    DO
      temp ← temp – 1;
      DEST ← temp;
    OD;
FI;

```

Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

Protected Mode Exceptions

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

BSR—Bit Scan Reverse (Continued)

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

BSWAP—Byte Swap

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

Description

Reverses the byte order of a 32-bit (destination) register: bits 0 through 7 are swapped with bits 24 through 31, and bits 8 through 15 are swapped with bits 16 through 23. This instruction is provided for converting little-endian values to big-endian format and vice versa.

To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

IA-32 Architecture Compatibility

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

Operation

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

Flags Affected

None.

Exceptions (All Operating Modes)

None.

BT—Bit Test

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

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

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

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

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

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

Operation

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

BT—Bit Test (Continued)

Flags Affected

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

BTC—Bit Test and Complement

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

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

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

Operation

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

$\text{Bit}(\text{BitBase}, \text{BitOffset}) \leftarrow \text{NOT } \text{Bit}(\text{BitBase}, \text{BitOffset});$

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.

BTC—Bit Test and Complement (Continued)

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

BTR—Bit Test and Reset

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

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

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

Operation

CF ← Bit(BitBase, BitOffset)
 Bit(BitBase, BitOffset) ← 0;

Flags Affected

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

Protected Mode Exceptions

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

BTR—Bit Test and Reset (Continued)

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

BTS—Bit Test and Set

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

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

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

Operation

CF ← Bit(BitBase, BitOffset)
 Bit(BitBase, BitOffset) ← 1;

Flags Affected

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

Protected Mode Exceptions

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

BTS—Bit Test and Set (Continued)

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

CALL—Call Procedure

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

Description

Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of calls:

- Near call—A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call—A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Inter-privilege-level far call—A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- Task switch—A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 6, *Task Management*, in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) onto the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified with the target operand. The target operand specifies either an absolute offset in the code segment (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register, which points to the instruction following the CALL instruction). The CS register is not changed on near calls.

CALL—Call Procedure (Continued)

For a near call, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, 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.

CALL—Call Procedure (Continued)

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

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. Here again, the target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.) On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an (optional) set of parameters from the calling procedure's stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction, is somewhat similar to executing a call through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset in the target operand is ignored.) The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into EIP register so that the task begins executing again at this next instruction.

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

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

CALL—Call Procedure (Continued)

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, the calls should be made through a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset is saved. Also, the call should be made using a 16-bit call gate so that 16-bit values will be pushed on the stack. See Chapter 17, *Mixing 17-Bit and 32-Bit Code*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information on making calls between 16-bit and 32-bit code segments.

Operation

IF near call

 THEN IF near relative call

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

 THEN IF OperandSize = 32

 THEN

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

 Push(EIP);

 EIP ← EIP + DEST; (* DEST is *rel32* *)

 ELSE (* OperandSize = 16 *)

 IF stack not large enough for a 2-byte return address THEN #SS(0); FI;

 Push(IP);

 EIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is *rel16* *)

 FI;

 FI;

 ELSE (* near absolute call *)

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

 IF OperandSize = 32

 THEN

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

 Push(EIP);

 EIP ← DEST; (* DEST is *r/m32* *)

 ELSE (* OperandSize = 16 *)

 IF stack not large enough for a 2-byte return address THEN #SS(0); FI;

 Push(IP);

 EIP ← DEST AND 0000FFFFH; (* DEST is *r/m16* *)

 FI;

 FI;

FI;

IF far call AND (PE = 0 OR (PE = 1 AND VM = 1)) (* real-address or virtual-8086 mode *)

 THEN

 IF OperandSize = 32

 THEN

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

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

CALL—Call Procedure (Continued)

```

        Push(CS); (* padded with 16 high-order bits *)
        Push(EIP);
        CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
        EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
    ELSE (* OperandSize = 16 *)
        IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
        IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
        Push(CS);
        Push(IP);
        CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
        EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16] *)
        EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
    FI;
FI;

IF far call AND (PE = 1 AND VM = 0) (* Protected mode, not virtual-8086 mode *)
    THEN
        IF segment selector in target operand null THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new code segment selector);
        FI;
        Read type and access rights of selected segment descriptor;
        IF segment type is not a conforming or nonconforming code segment, call gate,
            task gate, or TSS THEN #GP(segment selector); FI;
        Depending on type and access rights
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
    FI;

CONFORMING-CODE-SEGMENT:
    IF DPL > CPL THEN #GP(new code segment selector); FI;
    IF segment not present THEN #NP(new code segment selector); FI;
    IF OperandSize = 32
        THEN
            IF stack not large enough for a 6-byte return address THEN #SS(0); FI;
            IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
            Push(CS); (* padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST[NewCodeSegmentSelector];
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL
            EIP ← DEST[offset];
        
```

CALL—Call Procedure (Continued)

```

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

```

NONCONFORMING-CODE-SEGMENT:

```

IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(new code segment selector); FI;
IF segment not present THEN #NP(new code segment selector); FI;
IF stack not large enough for return address THEN #SS(0); FI;
tempEIP ← DEST[offset]
IF OperandSize=16
    THEN
        tempEIP ← tempEIP AND 0000FFFFH; (* clear upper 16 bits *)
FI;
IF tempEIP outside code segment limit THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        Push(CS); (* padded with 16 high-order bits *)
        Push(EIP);
        CS ← DEST[NewCodeSegmentSelector];
        (* segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        EIP ← tempEIP;
    ELSE (* OperandSize = 16 *)
        Push(CS);
        Push(IP);
        CS ← DEST[NewCodeSegmentSelector];
        (* segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        EIP ← tempEIP;
FI;
END;

```

CALL-GATE:

```

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

```

CALL—Call Procedure (Continued)

```

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

```

MORE-PRIVILEGE:

```

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

```

CALL—Call Procedure (Continued)

```

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

SAME-PRIVILEGE:
    IF CallGateSize = 32
        THEN
            IF stack does not have room for 8 bytes
                THEN #SS(0); FI;
            IF EIP not within code segment limit then #GP(0); FI;
            CS:EIP ← CallGate(CS:EIP) (* segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* return address to calling procedure *)
        ELSE (* CallGateSize = 16 *)
            IF stack does not have room for 4 bytes
                THEN #SS(0); FI;
            IF IP not within code segment limit THEN #GP(0); FI;
            CS:IP ← CallGate(CS:instruction pointer)
            (* segment descriptor information also loaded *)
            Push(oldCS:oldIP); (* return address to calling procedure *)
        FI;
    CS(RPL) ← CPL
END;

```


CALL—Call Procedure (Continued)

TASK-GATE:

```

IF task gate DPL < CPL or RPL
    THEN #GP(task gate selector);
FI;
IF task gate not present
    THEN #NP(task gate selector);
FI;
Read the TSS segment selector in the task-gate descriptor;
IF TSS segment selector local/global bit is set to local
    OR index not within GDT limits
    THEN #GP(TSS selector);
FI;
Access TSS descriptor in GDT;

IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
    THEN #GP(TSS selector);
FI;
IF TSS not present
    THEN #NP(TSS selector);
FI;
SWITCH-TASKS (with nesting) to TSS;
IF EIP not within code segment limit
    THEN #GP(0);
FI;
END;
```

TASK-STATE-SEGMENT:

```

IF TSS DPL < CPL or RPL
    OR TSS descriptor indicates TSS not available
    THEN #GP(TSS selector);
FI;
IF TSS is not present
    THEN #NP(TSS selector);
FI;
SWITCH-TASKS (with nesting) to TSS;
IF EIP not within code segment limit
    THEN #GP(0);
FI;
END;
```

Flags Affected

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

CALL—Call Procedure (Continued)**Protected Mode Exceptions**

#GP(0)	<p>If target offset in destination operand is beyond the new code segment limit.</p> <p>If the segment selector in the destination operand is null.</p> <p>If the code segment selector in the gate is null.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</p>
#GP(selector)	<p>If code segment or gate or TSS selector index is outside descriptor table limits.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.</p> <p>If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.</p> <p>If the segment selector from a call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a call gate is greater than the CPL.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.</p> <p>If a memory operand effective address is outside the SS segment limit.</p>

CALL—Call Procedure (Continued)

#SS(selector)	<p>If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.</p> <p>If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.</p> <p>If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.</p>
#NP(selector)	<p>If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present.</p>
#TS(selector)	<p>If the new stack segment selector and ESP are beyond the end of the TSS.</p> <p>If the new stack segment selector is null.</p> <p>If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.</p> <p>If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.</p> <p>If the new stack segment is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>
#AC(0)	<p>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</p>

Real-Address Mode Exceptions

#GP	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the target offset is beyond the code segment limit.</p>
-----	--

Virtual-8086 Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the target offset is beyond the code segment limit.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>
#AC(0)	<p>If alignment checking is enabled and an unaligned memory reference is made.</p>

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

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

Description

Double the size of the source operand by means of sign extension (see Figure 7-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the higher 16 bits of the EAX register.

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

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

Operation

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

FI;

Flags Affected

None.

Exceptions (All Operating Modes)

None.

CDQ—Convert Double to Quad

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

CLC—Clear Carry Flag

Opcode	Instruction	Description
F8	CLC	Clear CF flag

Description

Clears the CF flag in the EFLAGS register.

Operation

$CF \leftarrow 0$;

Flags Affected

The CF flag is 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

$DF \leftarrow 0$;

Flags Affected

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

Exceptions (All Operating Modes)

None.

CLFLUSH—Flush Cache Line

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

Description

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

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

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, WT memory types). PREFETCH h instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH h instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

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

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

Operation

Flush_Cache_Line(SRC)

CLFLUSH—Cache Line Flush (Continued)

Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSH `void_mm_clflush(void const *p)`

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID feature flag CLFSH is 0.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

CLI — Clear Interrupt Flag

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

Description

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 effect 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.							

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

```

CLI — Clear Interrupt Flag (Continued)

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

$CR0(TS) \leftarrow 0;$

Flags Affected

The TS flag in CR0 register is cleared.

Protected Mode Exceptions

#GP(0)

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)

CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement CF flag

Description

Complements the CF flag in the EFLAGS register.

Operation

$CF \leftarrow \text{NOT } CF;$

Flags Affected

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

CMOVcc—Conditional Move

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

CMOV_{cc}—Conditional Move (Continued)

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

Description

The CMOV_{cc} instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV_{cc} instruction.

These instructions can move a 16- or 32-bit value from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The conditions for each CMOV_{cc} mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

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

CMOV_{cc}—Conditional Move (Continued)

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.

CMOV_{cc}—Conditional Move (Continued)

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

CMP—Compare Two Operands

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

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The CMP instruction is typically used in conjunction with a conditional jump (*Jcc*), condition move (*CMOVcc*), or *SETcc* instruction. The condition codes used by the *Jcc*, *CMOVcc*, and *SETcc* instructions are based on the results of a CMP instruction. Appendix B, *EFLAGS Condition Codes*, in the *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.

CMP—Compare Two Operands (Continued)

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

CMPPD—Compare Packed Double-Precision Floating-Point Values

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

Description

Performs a SIMD compare of the two packed double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The comparison predicate operand is an 8-bit immediate the first 3 bits of which define the type of comparison to be made (see Table 3-6); bits 4 through 7 of the immediate are reserved.

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

Predic- cate	imm8 Encod- ing	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emula- tion	Result if NaN Operand	QNaN Oper- and Signals Invalid
EQ	000B	Equal	$A = B$		False	No
LT	001B	Less-than	$A < B$		False	Yes
LE	010B	Less-than-or-equal	$A \leq B$		False	Yes
		Greater than	$A > B$	Swap Operands, Use LT	False	Yes
		Greater-than-or-equal	$A \geq B$	Swap Operands, Use LE	False	Yes
UNORD	011B	Unordered	$A, B = \text{Unordered}$		True	No
NEQ	100B	Not-equal	$A \neq B$		True	No
NLT	101B	Not-less-than	$\text{NOT}(A < B)$		True	Yes
NLE	110B	Not-less-than-or-equal	$\text{NOT}(A \leq B)$		True	Yes
		Not-greater-than	$\text{NOT}(A > B)$	Swap Operands, Use NLT	True	Yes
		Not-greater-than-or-equal	$\text{NOT}(A \geq B)$	Swap Operands, Use NLE	True	Yes
ORD	111B	Ordered	$A, B = \text{Ordered}$		False	No

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

The unordered relationship is true when at least one of the two source operands being compared is a NaN; 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.

Pseudo-Op	CMPPD Implementation
CMPEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 0
CMPLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 1
CMPLDPD 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.)

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

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.

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

CMPPS—Compare Packed Single-Precision Floating-Point Values

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

Description

Performs a SIMD compare of the four packed single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The comparison predicate operand is an 8-bit immediate the first 3 bits of which define the type of comparison to be made (see Table 3-6); 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:

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

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

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

Operation

CASE (COMPARISON PREDICATE) OF

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

EASC

```

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

```

Intel C/C++ Compiler Intrinsic Equivalents

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

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

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

SIMD Floating-Point Exceptions

Invalid if SNaN 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 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.

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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 the source operands are located in memory. The address of the first source operand is read from either the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the second source operand is read from either the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct **type** (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct **location**. The locations of the source operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI and ES:(E)DI registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), or CMPSD (doubleword comparison).

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

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 this chapter for a description of the REP prefix.

Operation

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

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

Flags Affected

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

CMPD—Compare Scalar Double-Precision Floating-Point Values

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

Description

Compares the low double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand; the high quadword remains unchanged. The comparison predicate operand is an 8-bit immediate the first 3 bits of which define the type of comparison to be made (see Table 3-6); 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 CMPD instruction.

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

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

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

Operation

CASE (COMPARISON PREDICATE) OF

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

DEFAULT: Reserved;

CMP0 ← DEST[63-0] OP SRC[63-0];

IF CMP0 = TRUE

THEN DEST[63-0] ← FFFFFFFFFFFFFFFFH

ELSE DEST[63-0] ← 0000000000000000H; FI;

* DEST[127-64] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalents

CMPSD for equality	<code>__m128d _mm_cmpeq_sd(__m128d a, __m128d b)</code>
CMPSD for less-than	<code>__m128d _mm_cmplt_sd(__m128d a, __m128d b)</code>
CMPSD for less-than-or-equal	<code>__m128d _mm_cmple_sd(__m128d a, __m128d b)</code>
CMPSD for greater-than	<code>__m128d _mm_cmpgt_sd(__m128d a, __m128d b)</code>
CMPSD for greater-than-or-equal	<code>__m128d _mm_cmpge_sd(__m128d a, __m128d b)</code>
CMPSD for inequality	<code>__m128d _mm_cmpneq_sd(__m128d a, __m128d b)</code>
CMPSD for not-less-than	<code>__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)</code>
CMPSD for not-greater-than	<code>__m128d _mm_cmpngt_sd(__m128d a, __m128d b)</code>
CMPSD for not-greater-than-or-equal	<code>__m128d _mm_cmpnge_sd(__m128d a, __m128d b)</code>
CMPSD for ordered	<code>__m128d _mm_cmpord_sd(__m128d a, __m128d b)</code>
CMPSD for unordered	<code>__m128d _mm_cmpunord_sd(__m128d a, __m128d b)</code>
CMPSD for not-less-than-or-equal	<code>__m128d _mm_cmpnle_sd(__m128d a, __m128d b)</code>

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

SIMD Floating-Point Exceptions

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

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

CMPSS—Compare Scalar Single-Precision Floating-Point Values

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

Description

Compares the low single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand; the 3 high-order doublewords remain unchanged. The comparison predicate operand is an 8-bit immediate the first 3 bits of which define the type of comparison to be made (see Table 3-6); 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.

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

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

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

Operation

CASE (COMPARISON PREDICATE) OF

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

DEFAULT: Reserved;

CMP0 ← DEST[31-0] OP SRC[31-0];

IF CMP0 = TRUE

THEN DEST[31-0] ← FFFFFFFFH

ELSE DEST[31-0] ← 00000000H; FI;

* DEST[127-32] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalents

CMPSS for equality	__m128 _mm_cmpeq_ss(__m128 a, __m128 b)
CMPSS for less-than	__m128 _mm_cmlt_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)

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

SIMD Floating-Point Exceptions

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

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

CMPXCHG—Compare and Exchange

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

CMPXCHG—Compare and Exchange (Continued)

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

CMPXCHG8B—Compare and Exchange 8 Bytes

Opcode	Instruction	Description
0F C7 /1 m64	CMPXCHG8B <i>m64</i>	Compare EDX:EAX with <i>m64</i> . If equal, set ZF and load ECX:EBX into <i>m64</i> . Else, clear ZF and load <i>m64</i> into EDX:EAX.

Description

Compares the 64-bit value in EDX:EAX with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX. The destination operand is an 8-byte memory location. For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

IA-32 Architecture Compatibility

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

Operation

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

Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

CMPXCHG8B—Compare and Exchange 8 Bytes (Continued)

Protected Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If the destination is located in a 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 <i>xmm1</i> , <i>xmm2/m64</i>	Compare low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.

Description

Compares the double-precision floating-point values in the low quadwords of source operand 1 (first operand) and source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

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

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

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

Operation

```

RESULT ← OrderedCompare(DEST[63-0] <> SRC[63-0]) {
* Set EFLAGS *CASE (RESULT) OF
    UNORDERED:    ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN:    ZF,PF,CF ← 001;
    EQUAL:        ZF,PF,CF ← 100;
ESAC;
OF,AF,SF ← 0;
    
```

Intel C/C++ Compiler Intrinsic Equivalents

```

int_mm_comieq_sd(__m128d a, __m128d b)
int_mm_comilt_sd(__m128d a, __m128d b)
int_mm_comile_sd(__m128d a, __m128d b)
int_mm_comigt_sd(__m128d a, __m128d b)
int_mm_comige_sd(__m128d a, __m128d b)
int_mm_comineq_sd(__m128d a, __m128d b)
    
```

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

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

Description

Compares the single-precision floating-point values in the low doublewords of source operand 1 (first operand) and the source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

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

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

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

Operation

```
RESULT ← OrderedCompare(SRC1[31-0] <> SRC2[31-0]) {
```

```
* Set EFLAGS *CASE (RESULT) OF
  UNORDERED:    ZF,PF,CF ← 111;
  GREATER_THAN: ZF,PF,CF ← 000;
  LESS_THAN:    ZF,PF,CF ← 001;
  EQUAL:        ZF,PF,CF ← 100;
```

```
ESAC;
OF,AF,SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalents

```
int_mm_comieq_ss(__m128 a, __m128 b)
int_mm_comilt_ss(__m128 a, __m128 b)
int_mm_comile_ss(__m128 a, __m128 b)
int_mm_comigt_ss(__m128 a, __m128 b)
int_mm_comige_ss(__m128 a, __m128 b)
int_mm_comineq_ss(__m128 a, __m128 b)
```


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

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

CPUID—CPU Identification

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

Description

Returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers. The information returned is selected by entering a value in the EAX register before the instruction is executed. Table 3-7 shows the information returned, depending on the initial value loaded into the EAX register.

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction.

The information returned with the CPUID instruction is divided into two groups: basic information and extended function information. Basic information is returned by entering an input value of from 0 to 3 in the EAX register depending on the IA-32 processor type; extended function information is returned by entering an input value of from 80000000H to 80000004H. The extended function CPUID information was introduced in the Pentium 4 processor and is not available in earlier IA-32 processors. Table 3-8 shows the maximum input value that the processor recognizes for the CPUID instruction for basic information and for extended function information, for each family of IA-32 processors on which the CPUID instruction is implemented.

If a higher value than is shown in Table 3-7 is entered for a particular processor, the information for the highest useful basic information value is returned. For example, if an input value of 5 is entered in EAX for a Pentium 4 processor, the information for an input value of 2 is returned. The exception to this rule is the input values that return extended function information (currently, the values 80000000H through 80000004H). For a Pentium 4 processor, entering an input value of 80000005H or above, returns the information for an input value of 2.

The CPUID instruction can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed (see “Serializing Instructions” in Chapter 7 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*).

When the input value in the EAX register is 0, the processor returns the highest value the CPUID instruction recognizes in the EAX register for returning basic CPUID information (see Table 3-8). A vendor identification string is returned in the EBX, EDX, and ECX registers. For Intel processors, the vendor identification string is “GenuineIntel” as follows:

```

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

```

CPUID—CPU Identification (Continued)

Table 3-7. 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 3-8). “Genu” “ntel” “inel”
1H	EAX EBX ECX EDX	Version Information (Type, Family, Model, and Stepping ID) Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size. (Value * 8 = cache line size in bytes) Bits 23-16: Number of logical processors per physical processor. Bits 31-24: Local APIC ID Extended Feature Information (see Figure 3-4 and Table 3-10) Feature Information (see Figure 3-5 and Table 3-11)
2H	EAX EBX ECX EDX	Cache and TLB Information Cache and TLB Information Cache and TLB Information Cache and TLB Information
3H	EAX EBX ECX EDX	Reserved. Reserved. Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	Extended Function CPUID Information	
80000000H	EAX EBX ECX EDX	Maximum Input Value for Extended Function CPUID Information (see Table 3-8). Reserved. Reserved. Reserved.
80000001H	EAX EBX ECX EDX	Extended Processor Signature and Extended Feature Bits. (Currently Reserved.) Reserved. Reserved. Reserved.
80000002H	EAX EBX ECX EDX	Processor Brand String. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000003H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.

CPUID—CPU Identification (Continued)

Table 3-7. Information Returned by CPUID Instruction (Contd.)

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

Table 3-8. 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
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

CPUID—CPU Identification (Continued)

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

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

The available processor types are given in Table 3-9. Intel releases information on stepping IDs as needed.

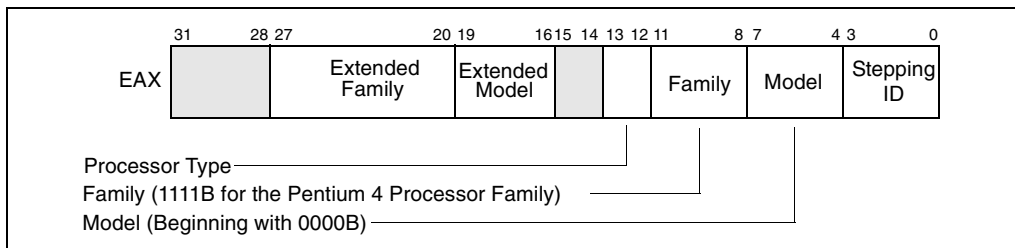


Figure 3-3. Version Information in the EAX Register

Table 3-9. Processor Type Field

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

NOTE:

* Not applicable to Intel486 processors.

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

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

CPUID—CPU Identification (Continued)

When the input value in EAX is 1, three unrelated pieces of information are returned to the EBX register:

- Brand index (low byte of EBX)—this number provides an entry into a brand string table that contains brand strings for IA-32 processors. See “Brand Identification” later in the description of this instruction for information about the intended use of brand indices. This field was introduced in the Pentium® III Xeon™ processors.
- CLFLUSH instruction cache line size (second byte of EBX)—this number indicates the size of the cache line flushed with CLFLUSH instruction in 8-byte increments. This field was introduced 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.

When the input value in EAX is 1, feature information is also returned in ECX and EDX. Figure 3-4 and Table 3-10 show encodings for the ECX register. Figure 3-5 and Table 3-11 show encodings for EDX. For all the feature flags currently returned in ECX and EDX, a 1 indicates that the feature is supported. Software should identify Intel as the vendor to properly interpret the feature flags. (Software should not depend on a 1 indicating the presence of a feature for future feature flags.)

CPUID—CPU Identification (Continued)

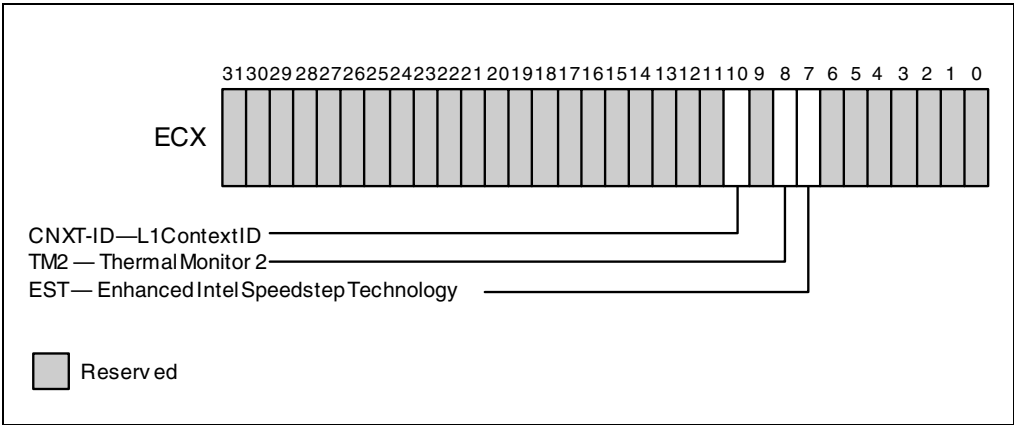


Figure 3-4. Extended Feature Flags Returned in ECX Register

Table 3-10. Extended Feature Flags Returned in ECX Register

Bit #	Mnemonic	Description
7	EST	Enhanced Intel® SpeedStep® Technology. A value of 1 indicates the processor supports the Enhanced Intel SpeedStep technology.
8	TM2	Thermal Monitor 2. A value of 1 indicates the processor supports the new Thermal Monitor 2 technology.
10	CNXT-ID	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 indicate this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for more details.

CPUID—CPU Identification (Continued)

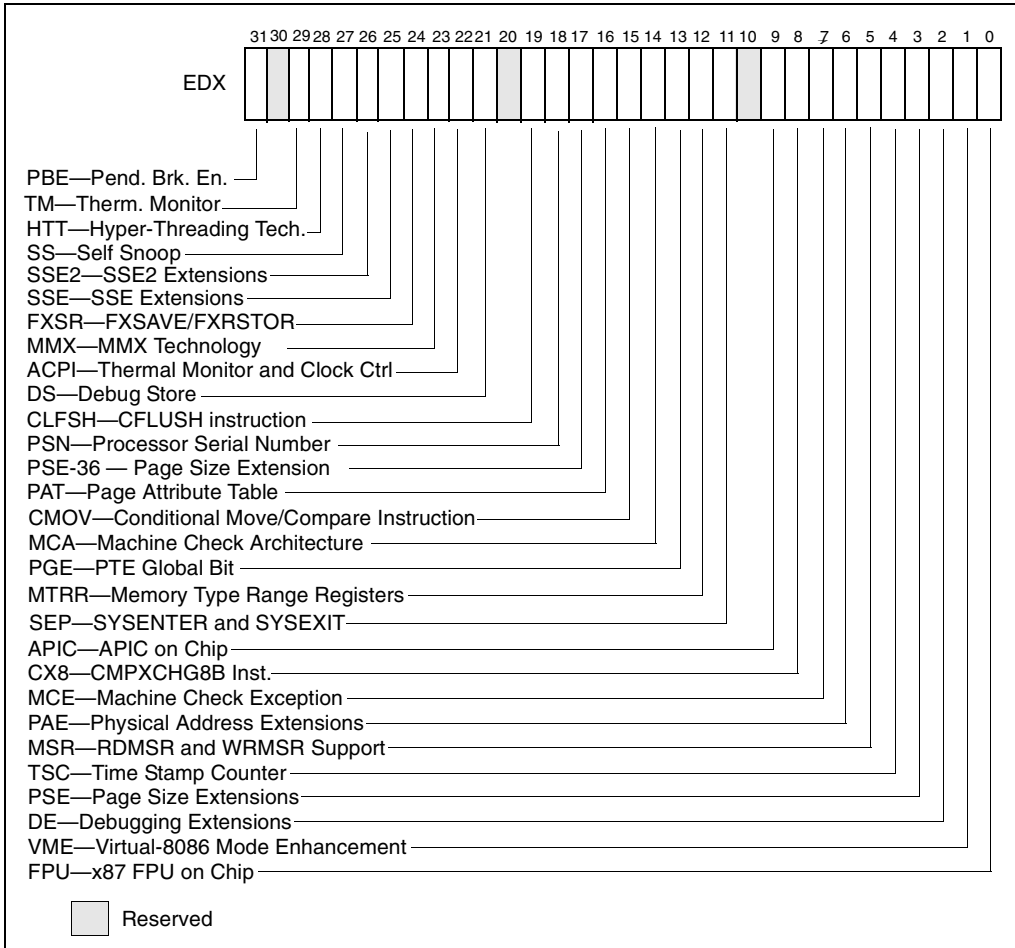


Figure 3-5. Feature Information in the EDX Register

CPUID—CPU Identification (Continued)

Table 3-11. CPUID Feature Flags Returned in EDX Register

Bit #	Mnemonic	Description
0	FPU	Floating Point Unit On-Chip. The processor contains an x87 FPU.
1	VME	Virtual 8086 Mode Enhancements. Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	Debugging Extensions. Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	Page Size Extension. Large pages of size 4Mbyte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	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.

CPUID—CPU Identification (Continued)

Table 3-11. CPUID Feature Flags Returned in EDX Register (Contd.)

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

CPUID—CPU Identification (Continued)

Table 3-11. CPUID Feature Flags Returned in 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.
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.

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

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs. The first member of the family of Pentium 4 processors will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 3-12 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.

CPUID—CPU Identification (Continued)
Table 3-12. Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4K-Byte Pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4M-Byte Pages, 4-way set associative, 2 entries
03H	Data TLB: 4K-Byte Pages, 4-way set associative, 64 entries
04H	Data TLB: 4M-Byte Pages, 4-way set associative, 8 entries
06H	1st-level instruction cache: 8K Bytes, 4-way set associative, 32 byte line size
08H	1st-level instruction cache: 16K Bytes, 4-way set associative, 32 byte line size
0AH	1st-level data cache: 8K Bytes, 2-way set associative, 32 byte line size
0CH	1st-level data cache: 16K Bytes, 4-way set associative, 32 byte line size
22H	3rd-level cache: 512K Bytes, 4-way set associative, 64 byte line size, 128 byte sector size
23H	3rd-level cache: 1M Bytes, 8-way set associative, 64 byte line size, 128 byte sector size
25H	3rd-level cache: 2M Bytes, 8-way set associative, 64 byte line size, 128 byte sector size
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: 128K Bytes, 4-way set associative, 32 byte line size
42H	2nd-level cache: 256K Bytes, 4-way set associative, 32 byte line size
43H	2nd-level cache: 512K Bytes, 4-way set associative, 32 byte line size
44H	2nd-level cache: 1M Byte, 4-way set associative, 32 byte line size
45H	2nd-level cache: 2M Byte, 4-way set associative, 32 byte line size
50H	Instruction TLB: 4-KByte and 2-MByte or 4-MByte pages, 64 entries
51H	Instruction TLB: 4-KByte and 2-MByte or 4-MByte pages, 128 entries
52H	Instruction TLB: 4-KByte and 2-MByte or 4-MByte pages, 256 entries
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

CPUID—CPU Identification (Continued)

Table 3-12. Encoding of Cache and TLB Descriptors (Contd.)

Descriptor Value	Cache or TLB Description
66H	1st-level data cache: 8KB, 4-way set associative, 64 byte line size
67H	1st-level data cache: 16KB, 4-way set associative, 64 byte line size
68H	1st-level data cache: 32KB, 4-way set associative, 64 byte line size
70H	Trace cache: 12K- μ op, 8-way set associative
71H	Trace cache: 16K- μ op, 8-way set associative
72H	Trace cache: 32K- μ op, 8-way set associative
78H	2nd-level cache: 1M Byte, 8-way set associative, 64byte line size
79H	2nd-level cache: 128KB, 8-way set associative, 64 byte line size, 128 byte sector size
7AH	2nd-level cache: 256KB, 8-way set associative, 64 byte line size, 128 byte sector size
7BH	2nd-level cache: 512KB, 8-way set associative, 64 byte line size, 128 byte sector size
7CH	2nd-level cache: 1MB, 8-way set associative, 64 byte line size, 128 byte sector size
7DH	2nd-level cache: 2M Byte, 8-way set associative, 64byte line size
82H	2nd-level cache: 256K Byte, 8-way set associative, 32 byte line size
83H	2nd-level cache: 512K Byte, 8-way set associative, 32 byte line size
84H	2nd-level cache: 1M Byte, 8-way set associative, 32 byte line size
85H	2nd-level cache: 2M Byte, 8-way set associative, 32 byte line size
86H	2nd-level cache: 512K Byte, 4-way set associative, 64 byte line size
87H	2nd-level cache: 1M Byte, 8-way set associative, 64 byte line size
B0H	Instruction TLB: 4M-Byte Pages, 4-way set associative, 128 entries
B3H	Data TLB: 4M-Byte Pages, 4-way set associative, 128 entries

CPUID—CPU Identification (Continued)

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

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

These values are interpreted as follows:

- The least-significant byte (byte 0) of register EAX is set to 01H, indicating that the CPUID instruction needs to be executed only once with an input value of 2 to retrieve complete information about the processor's caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor contains the following:
 - 50H—A 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
 - 5BH—A 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
 - 66H—An 8-KByte 1st level data cache, 4-way set associative, with a 64-byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain null descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor contains the following:
 - 00H—Null descriptor.
 - 70H—A 12-KByte 1st level code cache, 4-way set associative, with a 64-byte cache line size.
 - 7AH—A 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
 - 00H—Null descriptor.

CPUID—CPU Identification (Continued)

Brand Identification

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

The brand index was added to the CPUID instruction with the Pentium III Xeon processor and will be included on all future IA-32 processors, including the Pentium 4 processors. The brand index provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associated with an ASCII brand identification string that identifies the official Intel family and model number of a processor (for example, “Intel Pentium III processor”).

When executed with a value of 1 in the EAX register, the CPUID instruction returns the brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Table 3-13 shows those brand indices that currently have processor brand identification strings associated with them.

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

Table 3-13. Mapping of Brand Indices and IA-32 Processor Brand Strings

Brand Index	Brand String
0H	This processor does not support the brand identification feature
01H	Intel® Celeron® processor†
02H	Intel® Pentium® III processor†
03H	Intel® Pentium® III Xeon™ processor; If processor signature = 000006B1h, then “Intel® Celeron® processor”
04H	Intel® Pentium® III processor
06H	Mobile Intel® Pentium® III processor-M
07H	Mobile Intel® Celeron® processor†
08H	Intel® Pentium® 4 processor
09H	Intel® Pentium® 4 processor
0AH	Intel® Celeron® processor†
0BH	Intel® Xeon™ processor; If processor signature = 00000F13h, then “Intel® Xeon™ processor MP”
0CH	Intel® Xeon™ processor MP
0EH	Mobile Intel® Pentium® 4 processor-M; If processor signature = 00000F13h, then “Intel® Xeon™ processor”
0FH	Mobile Intel® Celeron® processor†
13H	Mobile Intel® Celeron® processor†
16H	Intel® Pentium® M processor
17 – 255	Reserved for future processor

Note

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

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

CPUID—CPU Identification (Continued)

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

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

Table 3-14 shows the brand string that is returned by the first processor in the family of Pentium 4 processors.

NOTE

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

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

1. Execute the CPUID instruction with input value in EAX of 80000000H.
2. If $((\text{EAX_Return_Value}) \text{ AND } (80000000\text{H}) \neq 0)$ then the processor supports the extended CPUID functions and EAX contains the largest extended function input value supported.
3. If $\text{EAX_Return_Value} \geq 80000004\text{H}$, then the CPUID instruction supports the brand string feature.

CPUID—CPU Identification (Continued)

Table 3-14. Processor Brand String Returned with First Pentium 4 Processor

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H;	“ ” “ ” “ ” “nl ”
80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	“(let “P)R” “itne” “R(mu”
80000004H	EAX = 20342029H; EBX = 20555043H; ECX = 30303531H EDX = 007A484DH	“ 4)” “ UPC” “0051” “\0zHM”

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

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

IA-32 Architecture Compatibility

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

CPUID—CPU Identification (Continued)**Operation**

CASE (EAX) OF

EAX = 0:

EAX ← highest basic function input value understood by CPUID;

EBX ← Vendor identification string;

EDX ← Vendor identification string;

ECX ← Vendor identification string;

BREAK;

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 3-4 *)

EDX ← Feature flags; (* See Figure 3-5 *)

BREAK;

EAX = 2H:

EAX ← Cache and TLB information;

EBX ← Cache and TLB information;

ECX ← Cache and TLB information;

EDX ← Cache and TLB information;

BREAK;

EAX = 3H:

EAX ← Reserved;

EBX ← Reserved;

ECX ← ProcessorSerialNumber[31:0];

(* Pentium III processors only, otherwise reserved *)

EDX ← ProcessorSerialNumber[63:32];

(* Pentium III processors only, otherwise reserved *)

BREAK;

EAX = 80000000H:

EAX ← highest extended function input value understood by CPUID;

EBX ← Reserved;

ECX ← Reserved;

EDX ← Reserved;

BREAK;

CPUID—CPU Identification (Continued)

EAX = 80000001H:

EAX ← Extended Processor Signature and Feature Bits (*Currently Reserved*);

EBX ← Reserved;

ECX ← Reserved;

EDX ← Reserved;

BREAK;

EAX = 80000002H:

EAX ← Processor Name;

EBX ← Processor Name;

ECX ← Processor Name;

EDX ← Processor Name;

BREAK;

EAX = 80000003H:

EAX ← Processor Name;

EBX ← Processor Name;

ECX ← Processor Name;

EDX ← Processor Name;

BREAK;

EAX = 80000004H:

EAX ← Processor Name;

EBX ← Processor Name;

ECX ← Processor Name;

EDX ← Processor Name;

BREAK;

DEFAULT: (* EAX > highest value recognized by CPUID *)

EAX ← Reserved; (* undefined*)

EBX ← Reserved; (* undefined*)

ECX ← Reserved; (* undefined*)

EDX ← Reserved; (* undefined*)

BREAK;

ESAC;

Flags Affected

None.

Exceptions (All Operating Modes)

None.

NOTE

In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.

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

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

Description

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTDQ2PD    __m128d _mm_cvtepi32_pd(__m128di a)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

CVTQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values (Continued)

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

Description

Converts four packed signed doubleword integers in the source operand (second operand) to four packed single-precision floating-point values in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. When a conversion is inexact, rounding is performed according to the rounding control bits in the MXCSR register.

Operation

```
DEST[31-0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31-0]);
DEST[63-32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63-32]);
DEST[95-64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95-64]);
DEST[127-96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127-96]);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTDQ2PS    __m128d __mm_cvtepi32_ps(__m128di a)
```

SIMD Floating-Point Exceptions

Precision.

Protected Mode Exceptions

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

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

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

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand, and the high quadword is cleared to all 0s. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

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

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand). The source operand can be an MMX 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.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand). The source operand can be an MMX 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.

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

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the 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 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.

CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers (Continued)

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

```
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.

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

If CPUID feature flag SSE2 is 0.

#AC(0)

If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

Interrupt 13

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

#NM

If TS in CR0 is set.

#XM

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

#UD

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)

For a page fault.

#AC(0)

If alignment checking is enabled and an unaligned memory reference is made.

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

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

Description

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

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

#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the 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.

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

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

Description

Converts a double-precision floating-point value in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand, and the upper 3 doublewords are left unchanged. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Operation

DEST[31-0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63-0]);
 * DEST[127-32] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalent

CVTSD2SS __m128_mm_cvtsd_ss(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set.

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

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

Description

Converts a signed doubleword integer in the source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a 32-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged.

Operation

DEST[63-0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31-0]);
 * DEST[127-64] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalent

int_mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE2 is 0.

CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value (Continued)

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

Description

Converts a signed doubleword integer in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a 32-bit memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Operation

DEST[31-0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31-0]);
 * DEST[127-32] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalent

`__m128_mm_cvtsi32_ss(__m128d a, int b)`

SIMD Floating-Point Exceptions

Precision.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0.

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

If CPUID feature flag SSE is 0.

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

Description

Converts a single-precision floating-point value in the source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand, and the high quadword is left unchanged.

Operation

DEST[63-0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31-0]);
 * DEST[127-64] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalent

CVTSS2SD __m128d_mm_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.

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

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the 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.

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

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Opcode	Instruction	Description
66 0F 2C /r	CVTTPD2PI <i>mm, xmm/m128</i>	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> using truncation.

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #SS(0) For an illegal address in the SS segment.

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

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

Opcode	Instruction	Description
66 0F E6	CVTTPD2DQ <i>xmm1</i> , <i>xmm2/m128</i>	Convert two packed double-precision floating-point values from <i>xmm2/m128</i> to two packed signed doubleword integers in <i>xmm1</i> using truncation.

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

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the 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 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.

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

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

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the 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 memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

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

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

#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

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

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the 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_cvttssd_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.

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the 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_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.

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

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

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

Description

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

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

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

Operation

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

Flags Affected

None.

Exceptions (All Operating Modes)

None.

CWDE—Convert Word to Doubleword

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

DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Description
27	DAA	Decimal adjust AL after addition

Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

Operation

```
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
```

DAA—Decimal Adjust AL after Addition (Continued)

Flags Affected

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

Exceptions (All Operating Modes)

None.

DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Description
2F	DAS	Decimal adjust AL after subtraction

Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

Operation

```

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 (OSZAPC)=010111
DAA             Before: AL=EEH  BL=47H  EFLAGS (OSZAPC)=010111
                After:  AL=88H  BL=47H  EFLAGS (OSZAPC)=X10111

```

DAS—Decimal Adjust AL after Subtraction (Continued)

Flags Affected

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

Exceptions (All Operating Modes)

None.

DEC—Decrement by 1

Opcode	Instruction	Description
FE /1	DEC <i>r/m8</i>	Decrement <i>r/m8</i> by 1
FF /1	DEC <i>r/m16</i>	Decrement <i>r/m16</i> by 1
FF /1	DEC <i>r/m32</i>	Decrement <i>r/m32</i> by 1
48+rw	DEC <i>r16</i>	Decrement <i>r16</i> by 1
48+rd	DEC <i>r32</i>	Decrement <i>r32</i> by 1

Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

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

Operation

DEST ← DEST – 1;

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

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

DEC—Decrement by 1 (Continued)

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

DIV—Unsigned Divide

Opcode	Instruction	Description
F6 /6	DIV <i>r/m8</i>	Unsigned divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder
F7 /6	DIV <i>r/m16</i>	Unsigned divide DX:AX by <i>r/m16</i> , with result stored in AX ← Quotient, DX ← Remainder
F7 /6	DIV <i>r/m32</i>	Unsigned divide EDX:EAX by <i>r/m32</i> , with result stored in EAX ← Quotient, EDX ← Remainder

Description

Divides (unsigned) the value in the AX, DX:AX, or EDX:EAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor), as shown in the following table:

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

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

Operation

```

IF SRC = 0
  THEN #DE; (* divide error *)
FI;
IF OperandSize = 8 (* word/byte operation *)
  THEN
    temp ← AX / SRC;
    IF temp > FFH
      THEN #DE; (* divide error *) ;
    ELSE
      AL ← temp;
      AH ← AX MOD SRC;
    FI;
  FI;

```

DIV—Unsigned Divide (Continued)

```

ELSE
    IF OperandSize = 16 (* doubleword/word operation *)
        THEN
            temp ← DX:AX / SRC;

            IF temp > FFFFH
                THEN #DE; (* divide error *) ;
            ELSE
                AX ← temp;
                DX ← DX:AX MOD SRC;
            FI;
        ELSE (* quadword/doubleword operation *)
            temp ← EDX:EAX / SRC;
            IF temp > FFFFFFFFH
                THEN #DE; (* divide error *) ;
            ELSE
                EAX ← temp;
                EDX ← EDX:EAX MOD SRC;
            FI;
        FI;
    FI;
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.

DIV—Unsigned Divide (Continued)

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 <i>xmm1</i> , <i>xmm2/m128</i>	Divide packed double-precision floating-point values in <i>xmm1</i> by packed double-precision floating-point values <i>xmm2/m128</i> .

Description

Performs a SIMD divide of the *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 a SIMD double-precision floating-point operation.

Operation

DEST[63-0] ← DEST[63-0] / (SRC[63-0]);
 DEST[127-64] ← DEST[127-64] / (SRC[127-64]);

Intel C/C++ Compiler Intrinsic Equivalent

DIVPD __m128 _mm_div_pd(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Protected Mode Exceptions

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

DIVPD—Divide Packed Double-Precision Floating-Point Values (Continued)

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
OF 5E /r	DIVPS <i>xmm1</i> , <i>xmm2/m128</i>	Divide packed single-precision floating-point values in <i>xmm1</i> by packed single-precision floating-point values <i>xmm2/m128</i> .

Description

Performs a SIMD divide of the two packed single-precision floating-point values in the destination operand (first operand) by the two packed single-precision floating-point values in the source operand (second operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a SIMD single-precision floating-point operation.

Operation

$$\begin{aligned} \text{DEST}[31-0] &\leftarrow \text{DEST}[31-0] / (\text{SRC}[31-0]); \\ \text{DEST}[63-32] &\leftarrow \text{DEST}[63-32] / (\text{SRC}[63-32]); \\ \text{DEST}[95-64] &\leftarrow \text{DEST}[95-64] / (\text{SRC}[95-64]); \\ \text{DEST}[127-96] &\leftarrow \text{DEST}[127-96] / (\text{SRC}[127-96]); \end{aligned}$$

Intel C/C++ Compiler Intrinsic Equivalent

DIVPS `__m128 _mm_div_ps(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Protected Mode Exceptions

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

DIVPS—Divide Packed Single-Precision Floating-Point Values (Continued)

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

DIVSD—Divide Scalar Double-Precision Floating-Point Values

Opcode	Instruction	Description
F2 0F 5E /r	DIVSD <i>xmm1</i> , <i>xmm2/m64</i>	Divide low double-precision floating-point value <i>n xmm1</i> by low double-precision floating-point value in <i>xmm2/mem64</i> .

Description

Divides the low double-precision floating-point value in the destination operand (first operand) by the low double-precision floating-point value in the source operand (second operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

Operation

DEST[63-0] ← DEST[63-0] / SRC[63-0];
 * DEST[127-64] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalent

DIVSD __m128d _mm_div_sd (m128d a, m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Protected Mode Exceptions

- #GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
- #SS(0) For an illegal address in the SS segment.
- #PF(fault-code) For a page fault.
- #NM If TS in CR0 is set.
- #XM If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
- #UD If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
 If EM in CR0 is set.
 If OSFXSR in CR4 is 0.
 If CPUID feature flag SSE2 is 0.

DIVSD—Divide Scalar Double-Precision Floating-Point Values (Continued)

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 5E /r	DIVSS <i>xmm1</i> , <i>xmm2/m32</i>	Divide low single-precision floating-point value in <i>xmm1</i> by low single-precision floating-point value in <i>xmm2/m32</i>

Description

Divides the low single-precision floating-point value in the destination operand (first operand) by the low single-precision floating-point value in the source operand (second operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

Operation

DEST[31-0] ← DEST[31-0] / SRC[31-0];

* DEST[127-32] remains unchanged *;

Intel C/C++ Compiler Intrinsic Equivalent

DIVSS __m128 _mm_div_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0. If EM in CR0 is set. If OSFXSR in CR4 is 0. If CPUID feature flag SSE is 0.

DIVSS—Divide Scalar Single-Precision Floating-Point Values (Continued)

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

Real-Address Mode Exceptions

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

#NM If TS in CR0 is set.

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

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

If EM in CR0 is set.

If OSFXSR in CR4 is 0.

If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

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

EMMS—Empty MMX 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 <i>imm16</i> ,0	Create a stack frame for a procedure
C8 iw 01	ENTER <i>imm16</i> ,1	Create a nested stack frame for a procedure
C8 iw ib	ENTER <i>imm16</i> , <i>imm8</i>	Create a nested stack frame for a procedure

Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits) or EBP (32 bits) register specifies the current frame pointer and whether SP (16 bits) or ESP (32 bits) specifies the stack pointer.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the EBP register onto the stack, copies the current stack pointer from the ESP register into the EBP register, and loads the ESP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See “Procedure Calls for Block-Structured Languages” in Chapter 6 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for more information about the actions of the ENTER instruction.

Operation

NestingLevel ← NestingLevel MOD 32

IF StackSize = 32

THEN

Push(EBP) ;

FrameTemp ← ESP;

ELSE (* StackSize = 16*)

Push(BP);

FrameTemp ← SP;

FI;

IF NestingLevel = 0

THEN GOTO CONTINUE;

FI;

ENTER—Make Stack Frame for Procedure Parameters (Continued)

```

IF (NestingLevel > 0)
  FOR i ← 1 TO (NestingLevel – 1)
    DO
      IF OperandSize = 32
        THEN
          IF StackSize = 32
            EBP ← EBP – 4;
            Push([EBP]); (* doubleword push *)
          ELSE (* StackSize = 16*)
            BP ← BP – 4;
            Push([BP]); (* doubleword push *)
          FI;
        ELSE (* OperandSize = 16 *)
          IF StackSize = 32
            THEN
              EBP ← EBP – 2;
              Push([EBP]); (* word push *)
            ELSE (* StackSize = 16*)
              BP ← BP – 2;
              Push([BP]); (* word push *)
            FI;
          FI;
        OD;
      IF OperandSize = 32
        THEN
          Push(FrameTemp); (* doubleword push *)
        ELSE (* OperandSize = 16 *)
          Push(FrameTemp); (* word push *)
        FI;
      GOTO CONTINUE;
    FI;
  CONTINUE:
  IF StackSize = 32
    THEN
      EBP ← FrameTemp
      ESP ← EBP – Size;
    ELSE (* StackSize = 16*)
      BP ← FrameTemp
      SP ← BP – Size;
    FI;
  END;

```

Flags Affected

None.

ENTER—Make Stack Frame for Procedure Parameters (Continued)**Protected Mode Exceptions**

- #SS(0) If the new value of the SP or ESP register is outside the stack segment limit.
- #PF(fault-code) If a page fault occurs.

Real-Address Mode Exceptions

- #SS(0) If the new value of the SP or ESP register is outside the stack segment limit.

Virtual-8086 Mode Exceptions

- #SS(0) If the new value of the SP or ESP register is outside the stack segment limit.
- #PF(fault-code) If a page fault occurs.

F2XM1—Compute 2^x-1

Opcode	Instruction	Description
D9 F0	F2XM1	Replace ST(0) with $(2^{ST(0)} - 1)$

Description

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

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

ST(0) SRC	ST(0) DEST
-1.0 to -0	-0.5 to -0
-0	-0
$+0$	$+0$
$+0$ to $+1.0$	$+0$ to 1.0

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

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

Operation

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

FPU Flags Affected

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

F2XM1—Compute 2^x-1 (Continued)**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.

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

NOTE:

F Means finite floating-point value.

Operation

$$ST(0) \leftarrow |ST(0)|$$

FPU Flags Affected

C1 Set to 0 if stack underflow occurred; otherwise, 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 <i>m32fp</i>	Add <i>m32fp</i> to ST(0) and store result in ST(0)
DC /0	FADD <i>m64fp</i>	Add <i>m64fp</i> to ST(0) and store result in ST(0)
D8 C0+i	FADD ST(0), ST(i)	Add ST(0) to ST(i) and store result in ST(0)
DC C0+i	FADD ST(i), ST(0)	Add ST(i) to ST(0) and store result in ST(i)
DE C0+i	FADDP ST(i), ST(0)	Add ST(0) to ST(i), store result in ST(i), and pop the register stack
DE C1	FADDP	Add ST(0) to ST(1), store result in ST(1), and pop the register stack
DA /0	FIADD <i>m32int</i>	Add <i>m32int</i> to ST(0) and store result in ST(0)
DE /0	FIADD <i>m16int</i>	Add <i>m16int</i> to ST(0) and store result in ST(0)

Description

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

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

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

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

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

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

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

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

FADD/FADDP/FIADD—Add (Continued)

		DEST						
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
SRC	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	NaN
	$-F$ or $-I$	$-\infty$	$-F$	SRC	SRC	$\pm F$ or ± 0	$+\infty$	NaN
	-0	$-\infty$	DEST	-0	± 0	DEST	$+\infty$	NaN
	$+0$	$-\infty$	DEST	± 0	$+0$	DEST	$+\infty$	NaN
	$+F$ or $+I$	$-\infty$	$\pm F$ or ± 0	SRC	SRC	$+F$	$+\infty$	NaN
	$+\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

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

Operation

IF instruction is FIADD

THEN

DEST \leftarrow DEST + ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (* source operand is floating-point value *)

DEST \leftarrow DEST + SRC;

FI;

IF instruction = 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.

FADD/FADDP/FIADD—Add (Continued)

#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 <i>m80 dec</i>	Convert BCD value to floating-point and push onto the FPU stack.

Description

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of -0 .

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

Operation

TOP \leftarrow TOP $-$ 1;
 ST(0) \leftarrow ConvertToDoubleExtendedPrecisionFP(SRC);

FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, 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.

FBLD—Load Binary Coded Decimal (Continued)

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FBSTP—Store BCD Integer and Pop

Opcode	Instruction	Description
DF /6	FBSTP m80bcd	Store ST(0) in m80bcd and pop ST(0).

Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

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

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	-D
$-1 < F < -0$	**
-0	-0
+0	+0
$+0 < F < +1$	**
$F \geq +1$	+D
$+\infty$ 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-arithmic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmic-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.

FBSTP—Store BCD Integer and Pop (Continued)**Operation**

DEST ← BCD(ST(0));
PopRegisterStack;

FPU Flags Affected

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

Floating-Point Exceptions

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

Protected Mode Exceptions

#GP(0)	If a segment register is being loaded with a segment selector that points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

FBSTP—Store BCD Integer and Pop (Continued)

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FCFS—Change Sign

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

Description

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

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

NOTE:

F Means finite floating-point value.

Operation

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.

NOTE:

* See “IA-32 Architecture Compatibility” below.

Description

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

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions 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.

FCLEX/FNCLEX—Clear Exceptions (Continued)

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual-8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FCMOV 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 conditions for each mnemonic are given in the Description column above and in Table 7-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOV 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 FCMOV cc 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.

FCMOV cc —Floating-Point Conditional Move (Continued)

Floating-Point Exceptions

#IS Stack underflow occurred.

Integer Flags Affected

None.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual-8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FCOM/FCOMP/FCOMPP—Compare Floating Point Values

Opcode	Instruction	Description
D8 /2	FCOM <i>m32fp</i>	Compare ST(0) with <i>m32fp</i> .
DC /2	FCOM <i>m64fp</i>	Compare ST(0) with <i>m64fp</i> .
D8 D0+i	FCOM ST(i)	Compare ST(0) with ST(i).
D8 D1	FCOM	Compare ST(0) with ST(1).
D8 /3	FCOMP <i>m32fp</i>	Compare ST(0) with <i>m32fp</i> and pop register stack.
DC /3	FCOMP <i>m64fp</i>	Compare ST(0) with <i>m64fp</i> and pop register stack.
D8 D8+i	FCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Compare ST(0) with ST(1) and pop register stack twice.

Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that -0.0 is equal to $+0.0$.

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.

