intel®

Intel Architecture Software Developer's Manual

Volume 1: Basic Architecture

NOTE: The *Intel Architecture Software Developer's Manual* consists of three books: *Basic Architecture*, Order Number 243190; *Instruction Set Reference Manual*, Order Number 243191; and the *System Programming Guide*, Order Number 243192.
Please refer to all three volumes when evaluating your design needs.

1997

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

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

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

Intel's Intel Architecture processors (e.g., Pentium® processor, Pentium processor with MMX™ technology, and Pentium Pro processor) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Such errata are not covered by Intel's warranty. Current characterized errata are available on request.

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

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

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

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

Copyright © Intel Corporation 1996, 1997.

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

TABLE OF CONTENTS

PAGE

CHAPTER 1
ABOUT THIS MANUAL
1.1. OVERVIEW OF THE *INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 1*: BASIC ARCHITECTURE 1-1
1.2. OVERVIEW OF THE *INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 2*: INSTRUCTION SET REFERENCE 1-2
1.3. OVERVIEW OF THE *INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 3*: SYSTEM PROGRAMMING GUIDE 1-3
1.4. NOTATIONAL CONVENTIONS 1-5
1.4.1. Bit and Byte Order 1-5
1.4.2. Reserved Bits and Software Compatibility 1-5
1.4.3. Instruction Operands 1-6
1.4.4. Hexadecimal and Binary Numbers 1-6
1.4.5. Segmented Addressing 1-7
1.4.6. Exceptions 1-7
1.5. RELATED LITERATURE 1-7

CHAPTER 2
INTRODUCTION TO THE INTEL ARCHITECTURE
2.1. BRIEF HISTORY OF THE INTEL ARCHITECTURE 2-1
2.2. INCREASING INTEL ARCHITECTURE PERFORMANCE AND MOORE'S LAW . . 2-4
2.3. BRIEF HISTORY OF THE INTEL ARCHITECTURE FLOATING-POINT UNIT. 2-5
2.4. INTRODUCTION TO THE PENTIUM® PRO PROCESSOR'S ADVANCED MICROARCHITECTURE 2-5
2.5. DETAILED DESCRIPTION OF THE PENTIUM® PRO PROCESSOR MICROARCHITECTURE 2-8
2.5.1. Memory Subsystem 2-8
2.5.2. The Fetch/Decode Unit 2-10
2.5.3. Instruction Pool (Reorder Buffer) 2-10
2.5.4. Dispatch/Execute Unit 2-11
2.5.5. Retirement Unit 2-12

CHAPTER 3
BASIC EXECUTION ENVIRONMENT
3.1. MODES OF OPERATION 3-1
3.2. OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT 3-2
3.3. MEMORY ORGANIZATION 3-2
3.4. MODES OF OPERATION 3-4
3.5. 32-BIT VS. 16-BIT ADDRESS AND OPERAND SIZES 3-5
3.6. REGISTERS 3-5
3.6.1. General-Purpose Data Registers 3-5
3.6.2. Segment Registers 3-7
3.6.3. EFLAGS Register 3-10
3.6.3.1. Status Flags 3-11
3.6.3.2. DF Flag 3-12
3.6.4. System Flags and IOPL Field 3-13
3.7. INSTRUCTION POINTER 3-14
3.8. OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES 3-14

PAGE

CHAPTER 4
PROCEDURE CALLS, INTERRUPTS, AND EXCEPTIONS

4.1. PROCEDURE CALL TYPES . . . 4-1
4.2. STACK . . . 4-1
4.2.1. Setting Up a Stack . . . 4-2
4.2.2. Stack Alignment . . . 4-3
4.2.3. Address-Size Attributes for Stack Accesses . . . 4-3
4.2.4. Procedure Linking Information . . . 4-3
4.2.4.1. Stack-Frame Base Pointer . . . 4-4
4.2.4.2. Return Instruction Pointer . . . 4-4
4.3. CALLING PROCEDURES USING CALL AND RET . . . 4-4
4.3.1. Near CALL and RET Operation . . . 4-5
4.3.2. Far CALL and RET Operation . . . 4-6
4.3.3. Parameter Passing . . . 4-6
4.3.3.1. Passing Parameters Through the General-Purpose Registers . . . 4-6
4.3.3.2. Passing Parameters on the Stack . . . 4-6
4.3.3.3. Passing Parameters in an Argument List . . . 4-7
4.3.4. Saving Procedure State Information . . . 4-7
4.3.5. Calls to Other Privilege Levels . . . 4-7
4.3.6. CALL and RET Operation Between Privilege Levels . . . 4-9
4.4. INTERRUPTS AND EXCEPTIONS . . . 4-10
4.4.1. Call and Return Operation for Interrupt or Exception Handling Procedures . . . 4-11
4.4.2. Calls to Interrupt or Exception Handler Tasks . . . 4-14
4.4.3. Interrupt and Exception Handling in Real-Address Mode . . . 4-15
4.4.4. INT n, INTO, INT 3, and BOUND Instructions . . . 4-15
4.5. PROCEDURE CALLS FOR BLOCK-STRUCTURED LANGUAGES . . . 4-16
4.5.1. ENTER Instruction . . . 4-16
4.5.2. LEAVE Instruction . . . 4-21

CHAPTER 5
DATA TYPES AND ADDRESSING MODES

5.1. FUNDAMENTAL DATA TYPES . . . 5-1
5.1.1. Alignment of Words, Doublewords, and Quadwords . . . 5-1
5.2. NUMERIC, POINTER, BIT FIELD, AND STRING DATA TYPES . . . 5-2
5.2.1. Integers . . . 5-2
5.2.2. Unsigned Integers . . . 5-4
5.2.3. BCD Integers . . . 5-4
5.2.4. Pointers . . . 5-4
5.2.5. Bit Fields . . . 5-4
5.2.6. Strings . . . 5-4
5.2.7. Floating-Point Data Types . . . 5-5
5.2.8. MMX™ Technology Data Types . . . 5-5
5.3. OPERAND ADDRESSING . . . 5-5
5.3.1. Immediate Operands . . . 5-5
5.3.2. Register Operands . . . 5-5
5.3.3. Memory Operands . . . 5-6
5.3.3.1. Specifying a Segment Selector . . . 5-6
5.3.3.2. Specifying an Offset . . . 5-7
5.3.3.3. Assembler and Compiler Addressing Modes . . . 5-9
5.3.4. I/O Port Addressing . . . 5-10

PAGE

CHAPTER 6
INSTRUCTION SET SUMMARY

6.1. NEW INTEL ARCHITECTURE INSTRUCTIONS 6-1
6.1.1. New Instructions Introduced with the MMX™ Technology 6-1
6.1.2. New Instructions in the Pentium® Pro Processor 6-1
6.1.3. New Instructions in the Pentium® Processor 6-2
6.1.4. New Instructions in the Intel486™ Processor 6-2
6.2. INSTRUCTION SET LIST 6-2
6.2.1. Integer Instructions 6-3
6.2.1.1. Data Transfer Instructions. 6-3
6.2.1.2. Binary Arithmetic Instructions 6-4
6.2.1.3. Decimal Arithmetic 6-4
6.2.1.4. Logic Instructions 6-5
6.2.1.5. Shift and Rotate Instructions. 6-5
6.2.1.6. Bit and Byte Instructions 6-5
6.2.1.7. Control Transfer Instructions. 6-6
6.2.1.8. String Instructions 6-7
6.2.1.9. Flag Control Instructions 6-8
6.2.1.10. Segment Register Instructions 6-8
6.2.1.11. Miscellaneous Instructions 6-9
6.2.2. MMX™ Technology Instructions 6-9
6.2.2.1. MMX™ Data Transfer Instructions 6-9
6.2.2.2. MMX™ Conversion Instructions 6-9
6.2.2.3. MMX™ Packed Arithmetic Instructions. 6-10
6.2.2.4. MMX™ Comparison Instructions 6-10
6.2.2.5. MMX™ Logic Instructions 6-10
6.2.2.6. MMX™ Shift and Rotate Instructions 6-11
6.2.2.7. MMX™ State Management. 6-11
6.2.3. Floating-Point Instructions 6-11
6.2.3.1. Data Transfer 6-11
6.2.3.2. Basic Arithmetic 6-12
6.2.3.3. Comparison. 6-13
6.2.3.4. Transcendental 6-13
6.2.3.5. Load Constants. 6-13
6.2.3.6. FPU Control 6-14
6.2.4. System Instructions 6-15
6.3. DATA MOVEMENT INSTRUCTIONS 6-16
6.3.1. General-Purpose Data Movement Instructions 6-16
6.3.1.1. Move Instruction 6-16
6.3.1.2. Conditional Move Instructions. 6-16
6.3.1.3. Exchange Instructions. 6-17
6.3.2. Stack Manipulation Instructions 6-19
6.3.2.1. Type Conversion Instructions 6-21
6.3.2.2. Simple Conversion 6-21
6.3.2.3. Move and Convert. 6-22
6.4. BINARY ARITHMETIC INSTRUCTIONS 6-22
6.4.1. Addition and Subtraction Instructions 6-22
6.4.2. Increment and Decrement Instructions 6-22
6.4.3. Comparison and Sign Change Instruction 6-23
6.4.4. Multiplication and Divide Instructions. 6-23
6.5. DECIMAL ARITHMETIC INSTRUCTIONS. 6-23
6.5.1. Packed BCD Adjustment Instructions 6-24

PAGE

6.5.2. Unpacked BCD Adjustment Instructions 6-24
6.6. LOGICAL INSTRUCTIONS 6-25
6.7. SHIFT AND ROTATE INSTRUCTIONS. 6-25
6.7.1. Shift Instructions 6-25
6.7.2. Double-Shift Instructions 6-27
6.7.3. Rotate Instructions. 6-27
6.8. BIT AND BYTE INSTRUCTIONS. 6-29
6.8.1. Bit Test and Modify Instructions 6-29
6.8.2. Bit Scan Instructions 6-29
6.8.3. Byte Set On Condition Instructions 6-29
6.8.4. Test Instruction 6-30
6.9. CONTROL TRANSFER INSTRUCTIONS 6-30
6.9.1. Unconditional Transfer Instructions 6-30
6.9.1.1. Jump Instruction 6-30
6.9.1.2. Call and Return Instructions 6-31
6.9.1.3. Return From Interrupt Instruction 6-31
6.9.2. Conditional Transfer Instructions. 6-31
6.9.2.1. Conditional Jump Instructions. 6-32
6.9.2.2. Loop Instructions 6-33
6.9.2.3. Jump If Zero Instructions 6-33
6.9.3. Software Interrupts 6-34
6.10. STRING OPERATIONS 6-34
6.10.1. Repeating String Operations 6-35
6.11. I/O INSTRUCTIONS. 6-36
6.12. ENTER AND LEAVE INSTRUCTIONS 6-36
6.13. EFLAGS INSTRUCTIONS 6-37
6.13.1. Carry and Direction Flag Instructions 6-37
6.13.2. Interrupt Flag Instructions 6-37
6.13.3. EFLAGS Transfer Instructions. 6-37
6.13.4. Interrupt Flag Instructions 6-38
6.14. SEGMENT REGISTER INSTRUCTIONS 6-38
6.14.1. Segment-Register Load and Store Instructions. 6-38
6.14.2. Far Control Transfer Instructions. 6-39
6.14.3. Software Interrupt Instructions. 6-39
6.14.4. Load Far Pointer Instructions 6-39
6.15. MISCELLANEOUS INSTRUCTIONS. 6-39
6.15.1. Address Computation Instruction 6-39
6.15.2. Table Lookup Instructions 6-40
6.15.3. Processor Identification Instruction 6-40
6.15.4. No-Operation and Undefined Instructions 6-40

CHAPTER 7
FLOATING-POINT UNIT

7.1. COMPATIBILITY AND EASE OF USE OF THE INTEL ARCHITECTURE FPU 7-1
7.2. REAL NUMBERS AND FLOATING-POINT FORMATS. 7-2
7.2.1. Real Number System 7-3
7.2.2. Floating-Point Format 7-3
7.2.2.1. Normalized Numbers 7-4
7.2.2.2. Biased Exponent. 7-5
7.2.3. Real Number and Non-number Encodings 7-5
7.2.3.1. Signed Zeros 7-6
7.2.3.2. Normalized and Denormalized Finite Numbers 7-6

PAGE

7.2.3.3. Signed Infinities. 7-8
7.2.3.4. NaNs. 7-8
7.2.4. Indefinite 7-8
7.3. FPU ARCHITECTURE 7-8
7.3.1. The FPU Data Registers 7-9
7.3.1.1. Parameter Passing With the FPU Register Stack. 7-11
7.3.2. FPU Status Register 7-12
7.3.2.1. Top of Stack (TOP) Pointer. 7-12
7.3.2.2. Condition Code Flags 7-12
7.3.2.3. Exception Flags 7-14
7.3.2.4. Stack Fault Flag 7-15
7.3.3. Branching and Conditional Moves on FPU Condition Codes 7-15
7.3.4. FPU Control Word 7-16
7.3.4.1. Exception-Flag Masks. 7-17
7.3.4.2. Precision Control Field 7-17
7.3.4.3. Rounding Control Field 7-18
7.3.5. Infinity Control Flag 7-20
7.3.6. FPU Tag Word. 7-20
7.3.7. The FPU Instruction and Operand (Data) Pointers 7-21
7.3.8. Last Instruction Opcode 7-21
7.3.9. Saving the FPU’s State 7-21
7.4. FLOATING-POINT DATA TYPES AND FORMATS. 7-24
7.4.1. Real Numbers 7-25
7.4.2. Binary Integers. 7-27
7.4.3. Decimal Integers 7-28
7.4.4. Unsupported Extended-Real Encodings 7-28
7.5. FPU INSTRUCTION SET. 7-29
7.5.1. Escape (ESC) Instructions. 7-30
7.5.2. FPU Instruction Operands 7-31
7.5.3. Data Transfer Instructions 7-31
7.5.4. Load Constant Instructions 7-33
7.5.5. Basic Arithmetic Instructions 7-33
7.5.6. Comparison and Classification Instructions. 7-34
7.5.6.1. Branching on the FPU Condition Codes 7-36
7.5.7. Trigonometric Instructions 7-37
7.5.8. Pi 7-37
7.5.9. Logarithmic, Exponential, and Scale 7-38
7.5.10. Transcendental Instruction Accuracy. 7-39
7.5.11. FPU Control Instructions 7-39
7.5.12. Waiting Vs. Non-waiting Instructions 7-40
7.5.13. Unsupported FPU Instructions. 7-41
7.6. OPERATING ON NANS 7-41
7.6.1. Uses for Signaling NANs 7-42
7.6.2. Uses for Quiet NANs 7-42
7.7. FLOATING-POINT EXCEPTION HANDLING 7-42
7.7.1. Arithmetic vs. Non-arithmetic Instructions 7-43
7.7.2. Automatic Exception Handling. 7-43
7.7.3. Software Exception Handling. 7-45
7.7.3.1. Native Mode 7-45
7.7.3.2. MS-DOS* Compatibility Mode. 7-45
7.7.3.3. Typical Floating-Point Exception Handler Actions 7-46
7.8. FLOATING-POINT EXCEPTION CONDITIONS 7-47

PAGE

7.8.1. Invalid Operation Exception. 7-47
7.8.1.1. Stack Overflow or Underflow Exception (#IS). 7-48
7.8.1.2. Invalid Arithmetic Operand Exception (#IA) 7-48
7.8.2. Divide-By-Zero Exception (#Z) 7-49
7.8.3. Denormal Operand Exception (#D) 7-50
7.8.4. Numeric Overflow Exception (#O) 7-50
7.8.5. Numeric Underflow Exception (#U) 7-52
7.8.6. Inexact-Result (Precision) Exception (#P). 7-53
7.8.7. Exception Priority. 7-53
7.9. FLOATING-POINT EXCEPTION SYNCHRONIZATION 7-54

CHAPTER 8
PROGRAMMING WITH THE INTEL MMX™ TECHNOLOGY

8.1. OVERVIEW OF THE MMX™ TECHNOLOGY PROGRAMMING ENVIRONMENT 8-1
8.1.1. MMX™ Registers 8-2
8.1.2. MMX™ Data Types 8-2
8.1.3. Single Instruction, Multiple Data (SIMD) Execution Model 8-3
8.1.4. Memory Data Formats. 8-4
8.1.5. Data Formats for MMX™ Registers 8-4
8.2. MMX™ INSTRUCTION SET 8-4
8.2.1. Saturation Arithmetic and Wraparound Mode 8-5
8.2.2. Instruction Operands 8-6
8.3. OVERVIEW OF THE MMX™ INSTRUCTION SET 8-6
8.3.1. Data Transfer Instructions 8-6
8.3.2. Arithmetic Instructions 8-8
8.3.2.1. Packed Addition And Subtraction 8-8
8.3.2.2. Packed Multiplication 8-8
8.3.2.3. Packed Multiply Add 8-8
8.3.3. Comparison Instructions 8-8
8.3.4. Conversion Instructions. 8-9
8.3.5. Logical Instructions 8-9
8.3.6. Shift Instructions 8-9
8.3.7. EMMS (Empty MMX™ State) Instruction 8-9
8.4. COMPATIBILITY WITH FPU ARCHITECTURE 8-10
8.4.1. MMX™ Instructions and the Floating-Point Tag Word 8-10
8.4.2. Effect of Instruction Prefixes on MMX™ Instructions 8-10
8.5. WRITING APPLICATIONS WITH MMX™ CODE 8-10
8.5.1. Detecting Support for MMX™ Technology Using the CPUID Instruction 8-11
8.5.2. Using the EMMS Instruction 8-11
8.5.3. Interfacing with MMX™ Code 8-12
8.5.4. Writing Code with MMX™ and Floating-Point Instructions 8-13
8.5.4.1. RECOMMENDATIONS AND GUIDELINES. 8-13
8.5.5. Using MMX™ Code in a Multitasking Operating System Environment. 8-14
8.5.5.1. COOPERATIVE MULTITASKING OPERATING SYSTEM 8-14
8.5.5.2. PREEMPTIVE MULTITASKING OPERATING SYSTEM. 8-14
8.5.6. Exception Handling in MMX™ Code 8-15
8.5.7. Register Mapping 8-15

CHAPTER 9
INPUT/OUTPUT

9.1. I/O PORT ADDRESSING. 9-1

PAGE

9.2. I/O PORT HARDWARE 9-1
9.3. I/O ADDRESS SPACE 9-2
9.3.1. Memory-Mapped I/O 9-2
9.4. I/O INSTRUCTIONS 9-3
9.5. PROTECTED-MODE I/O 9-4
9.5.1. I/O Privilege Level 9-4
9.5.2. I/O Permission Bit Map 9-5
9.6. ORDERING I/O 9-6

CHAPTER 10
PROCESSOR IDENTIFICATION AND FEATURE DETERMINATION

10.1. PROCESSOR IDENTIFICATION 10-1
10.2. IDENTIFICATION OF EARLIER INTEL ARCHITECTURE PROCESSORS 10-3

APPENDIX A
EFLAGS CROSS-REFERENCE

APPENDIX B
EFLAGS CONDITION CODES

APPENDIX C
FLOATING-POINT EXCEPTIONS SUMMARY

APPENDIX D
GUIDELINES FOR WRITING FPU EXCEPTION HANDLERS

D.1. ORIGIN OF THE MS-DOS* COMPATIBILITY MODE FOR HANDLING FPU EXCEPTIONS. D-2
D.2. IMPLEMENTATION OF THE MS-DOS* COMPATIBILITY MODE IN THE INTEL486™, PENTIUM®, AND PENTIUM PRO PROCESSORS D-3
D.2.1. MS-DOS* Compatibility Mode in the Intel486™ and Pentium® Processors D-3
D.2.1.1. Basic Rules: When FERR# Is Generated D-4
D.2.1.2. Recommended External Hardware to Support the MS-DOS* Compatibility Mode. D-5
D.2.1.3. No-Wait FPU Instructions Can Get FPU Interrupt in Window D-7
D.2.2. MS-DOS* Compatibility Mode in the Pentium® Pro Processor D-9
D.3. RECOMMENDED PROTOCOL FOR MS-DOS* COMPATIBILITY HANDLERS . . D-10
D.3.1. Floating-Point Exceptions and Their Defaults D-10
D.3.2. Two Options for Handling Numeric Exceptions D-11
D.3.2.1. Automatic Exception Handling: Using Masked Exceptions. D-11
D.3.2.2. Software Exception Handling D-13
D.3.3. Synchronization Required for Use of FPU Exception Handlers. D-14
D.3.3.1. Exception Synchronization: What, Why and When. D-14
D.3.3.2. Exception Synchronization Examples. D-15
D.3.3.3. Proper Exception Synchronization in General D-16
D.3.4. FPU Exception Handling Examples. D-16
D.3.5. Need for Storing State of IGNNE# Circuit If Using FPU and SMM D-20
D.3.6. Considerations When FPU Shared Between Tasks D-21
D.3.6.1. Speculatively Deferring FPU Saves, General Overview D-22
D.3.6.2. Tracking FPU Ownership D-22
D.3.6.3. interaction of FPU State Saves and Floating Point Exception Association. . D-23
D.3.6.4. Interrupt Routing From the Kernel D-26

PAGE

D.4. DIFFERENCES FOR HANDLERS USING NATIVE MODE. D-26
D.4.1. Origin With the Intel 286 and Intel 287, and Intel386™ and Intel 387 Processors D-27
D.4.2. Changes with Intel486™, Pentium“ and Pentium Pro Processors with CR0.NE=1 . . . . D-27
D.4.3. Considerations When FPU Shared Between Tasks Using Native Mode D-28

TABLE OF FIGURES

PAGE

Figure 1-1. Bit and Byte Order 1-5
Figure 2-1. The Processing Units in the Pentium® Pro Processor Microarchitecture and Their Interface with the Memory Subsystem 2-6
Figure 2-2. Functional Block Diagram of the Pentium® Pro Processor Microarchitecture . . 2-9
Figure 3-1. Pentium® Pro Processor Basic Execution Environment 3-2
Figure 3-2. Three Memory Management Models 3-3
Figure 3-3. Application Programming Registers 3-6
Figure 3-4. Alternate General-Purpose Register Names 3-7
Figure 3-5. Use of Segment Registers for Flat Memory Model. 3-8
Figure 3-6. Use of Segment Registers in Segmented Memory Model 3-9
Figure 3-7. EFLAGS Register 3-11
Figure 4-1. Stack Structure 4-2
Figure 4-2. Stack on Near and Far Calls. 4-5
Figure 4-3. Protection Rings 4-8
Figure 4-4. Stack Switch on a Call to a Different Privilege Level 4-9
Figure 4-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines . . . 4-13
Figure 4-6. Nested Procedures 4-18
Figure 4-7. Stack Frame after Entering the MAIN Procedure 4-19
Figure 4-8. Stack Frame after Entering Procedure A 4-19
Figure 4-9. Stack Frame after Entering Procedure B 4-20
Figure 4-10. Stack Frame after Entering Procedure C 4-21
Figure 5-1. Fundamental Data Types 5-1
Figure 5-2. Bytes, Words, Doublewords and Quadwords in Memory 5-2
Figure 5-3. Numeric, Pointer, and Bit Field Data Types 5-3
Figure 5-4. Memory Operand Address 5-6
Figure 5-5. Offset (or Effective Address) Computation 5-8
Figure 6-1. Operation of the PUSH Instruction 6-19
Figure 6-2. Operation of the PUSHA Instruction 6-20
Figure 6-3. Operation of the POP Instruction 6-20
Figure 6-4. Operation of the POPA Instruction 6-21
Figure 6-5. Sign Extension 6-21
Figure 6-6. SHL/SAL Instruction Operation. 6-25
Figure 6-7. SHR Instruction Operation 6-26
Figure 6-8. SAR Instruction Operation 6-26
Figure 6-9. SHLD and SHRD Instruction Operations 6-27
Figure 6-10. ROL, ROR, RCL, and RCR Instruction Operations 6-28
Figure 6-11. Flags Affected by the PUSHF, POPF, PUSHFD, and POPFD instructions . . 6-38
Figure 7-1. Binary Real Number System 7-3
Figure 7-2. Binary Floating-Point Format 7-4
Figure 7-3. Real Numbers and NaNs 7-6
Figure 7-4. Relationship Between the Integer Unit and the FPU 7-9
Figure 7-5. FPU Execution Environment. 7-10
Figure 7-6. FPU Data Register Stack 7-10
Figure 7-7. Example FPU Dot Product Computation 7-12
Figure 7-8. FPU Status Word 7-13
Figure 7-9. Moving the FPU Condition Codes to the EFLAGS Register. 7-16
Figure 7-10. FPU Control Word 7-17
Figure 7-11. FPU Tag Word 7-20
Figure 7-12. Contents of FPU Opcode Registers 7-22

PAGE

Figure 7-13. Protected Mode FPU State Image in Memory, 32-Bit Format 7-22
Figure 7-14. Real Mode FPU State Image in Memory, 32-Bit Format 7-23
Figure 7-15. Protected Mode FPU State Image in Memory, 16-Bit Format 7-23
Figure 7-16. Real Mode FPU State Image in Memory, 16-Bit Format 7-23
Figure 7-17. Floating-Point Unit Data Type Formats 7-24
Figure 8-1. MMX™ Register Set 8-2
Figure 8-2. MMX™ Data Types 8-3
Figure 8-3. Eight Packed Bytes in Memory (at address 1000H) 8-4
Figure 9-1. Memory-Mapped I/O 9-3
Figure 9-2. I/O Permission Bit Map 9-5
Figure D-1. Recommended Circuit for MS-DOS* Compatibility FPU Exception Handling. D-6
Figure D-2. Behavior of Signals During FPU Exception Handling D-7
Figure D-3. Timing of Receipt of External Interrupt D-8
Figure D-4. Arithmetic Example Using Infinity D-12
Figure D-5. General Program Flow for DNA Exception Handler D-25
Figure D-6. Program Flow for a Numeric Exception Dispatch Routine D-25

TABLE OF TABLES

PAGE

Table 2-1. Processor Performance Over Time and Other Key Features of the Intel Architecture 2-4
Table 3-1. Effective Operand- and Address-Size Attributes 3-15
Table 4-1. Exceptions and Interrupts 4-12
Table 5-1. Default Segment Selection Rules 5-7
Table 6-1. Move Instruction Operations 6-17
Table 6-2. Conditional Move Instructions 6-18
Table 6-3. Bit Test and Modify Instructions 6-29
Table 6-4. Conditional Jump Instructions 6-32
Table 6-5. Information Provided by the CPUID Instruction 6-40
Table 7-1. Real Number Notation 7-5
Table 7-2. Denormalization Process 7-7
Table 7-3. FPU Condition Code Interpretation 7-14
Table 7-4. Precision Control Field (PC) 7-17
Table 7-5. Rounding Control Field (RC) 7-18
Table 7-6. Rounding of Positive Numbers With Masked Overflow 7-19
Table 7-7. Rounding of Negative Numbers With Masked Overflow 7-19
Table 7-8. Length, Precision, and Range of FPU Data Types 7-25
Table 7-9. Real Number and NaN Encodings 7-26
Table 7-10. Binary Integer Encodings 7-27
Table 7-11. Packed Decimal Integer Encodings 7-29
Table 7-12. Unsupported Extended-Real Encodings 7-30
Table 7-13. Data Transfer Instructions 7-31
Table 7-14. Floating-Point Conditional Move Instructions 7-32
Table 7-15. Setting of FPU Condition Code Flags for Real Number Comparisons 7-35
Table 7-16. Setting of EFLAGS Status Flags for Real Number Comparisons 7-35
Table 7-17. TEST Instruction Constants for Conditional Branching 7-36
Table 7-18. Rules for Generating QNaNs 7-41
Table 7-19. Arithmetic and Non-arithmetic Instructions 7-44
Table 7-20. Invalid Arithmetic Operations and the Masked Responses to Them 7-49
Table 7-21. Divide-By-Zero Conditions and the Masked Responses to Them 7-50
Table 7-22. Masked Responses to Numeric Overflow 7-51
Table 8-1. Data Range Limits for Saturation 8-5
Table 8-2. MMX™ Instruction Set Summary 8-7
Table 8-3. Effect of Prefixes on MMX™ Instructions 8-10
Table 9-1. I/O Instruction Serialization 9-7
Table A-1. EFLAGS Cross-Reference A-1
Table B-1. EFLAGS Condition Codes B-1
Table C-1. Floating-Point Exceptions Summary C-1

intel®

1

About This Manual

CHAPTER 1 ABOUT THIS MANUAL

The *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture* (Order Number 243190) is part of a three-volume set that describes the architecture and programming environment of all Intel Architecture processors. The other two volumes in this set are:

- The *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference* (Order Number 243191).
- The *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide* (Order Number 243192).

The *Intel Architecture Software Developer's Manual, Volume 1,* describes the basic architecture and programming environment of an Intel Architecture processor; the *Intel Architecture Software Developer's Manual, Volume 2,* describes the instruction 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 *Intel Architecture Software Developer's Manual, Volume 3* describes the operating-system support environment of an Intel Architecture processor, including memory management, protection, task management, interrupt and exception handling, and system management mode. It also provides Intel Architecture processor compatibility information. This volume is aimed at operating-system and BIOS designers and programmers.

1.1. OVERVIEW OF THE *INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 1: BASIC ARCHITECTURE*

The contents of this manual are as follows:

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

Chapter 2 — Introduction to the Intel Architecture. Introduces the Intel Architecture and the families of Intel processors that are based on this architecture. It also gives an overview of the common features found in these processors and brief history of the Intel Architecture.

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

Chapter 4 — Procedure Calls, Interrupts, and Exceptions. Describes the procedure stack and the mechanisms provided for making procedure calls and for servicing interrupts and exceptions.

Chapter 5 — Data Types and Addressing Modes. Describes the data types and addressing modes recognized by the processor.

Chapter 6 — Instruction Set Summary. Gives an overview of all the Intel Architecture instructions except those executed by the processor's floating-point unit. The instructions are presented in functionally related groups.

Chapter 7 — Floating-Point Unit. Describes the Intel Architecture floating-point unit, including the floating-point registers and data types; gives an overview of the floating-point instruction set; and describes the processor's floating-point exception conditions.

Chapter 8 — Programming with Intel MMX™ Technology. Describes the Intel MMX™ technology, including MMX registers and data types, and gives an overview of the MMX instruction set.

Chapter 9 — Input/Output. Describes the processor's I/O architecture, including I/O port addressing, the I/O instructions, and the I/O protection mechanism.

Chapter 10 — Processor Identification and Feature Determination. Describes how to determine the CPU type and the features that are available in the processor.

Appendix A — EFLAGS Cross-Reference. Summaries how the Intel Architecture instructions affect the flags in the EFLAGS register.

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

Appendix C — Floating-Point Exceptions Summary. Summarizes the exceptions that can be raised by floating-point instructions.

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

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

The contents of the *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 *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 Intel Architecture 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 Intel Architecture 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 FPU and MMX instructions are included in this chapter.

Appendix A — Opcode Map. Gives an opcode map for the Intel Architecture instruction set.

Appendix B — Instruction Formats and Encodings. Gives the binary encoding of each form of each Intel Architecture instruction.

1.3. OVERVIEW OF THE *INTEL ARCHITECTURE SOFTWARE DEVELOPER'S MANUAL, VOLUME 3*: *SYSTEM PROGRAMMING GUIDE*

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

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

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

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

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

Chapter 6 — Task Management. Describes the mechanisms the Intel Architecture provides to support multitasking and inter-task protection.

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

Chapter 8 — Processor Management and Initialization. Defines the state of an Intel Architecture processor and its floating-point unit after reset initialization. This chapter also explains

how to set up an Intel Architecture processor for real-address mode operation and protected mode operation, and how to switch between modes.

Chapter 9 — Memory Cache Control. Describes the general concept of caching and the caching mechanisms supported by the Intel Architecture. This chapter also describes the memory type range registers (MTRRs) and how they can be used to map memory types of physical memory. MTRRs were introduced into the Intel Architecture with the Pentium® Pro processor.

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

Chapter 11 — System Management Mode (SMM). Describes the Intel Architecture's system management mode (SMM), which can be used to implement power management functions.

Chapter 12 — Machine Check Architecture. Describes the machine check architecture, which was introduced into the Intel Architecture with the Pentium processor.

Chapter 13 — Code Optimization. Discusses general optimization techniques for programming an Intel Architecture processor.

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

Chapter 15 — 8086 Emulation. Describes the real-address and virtual-8086 modes of the Intel Architecture.

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

Chapter 17 — Intel Architecture Compatibility. Describes the programming differences between the Intel 286, Intel386™, Intel486™, Pentium, and Pentium Pro processors. The differences among the 32-bit Intel Architecture processors (the Intel386, Intel486, Pentium, and Pentium Pro processors) are described throughout the three volumes of the *Intel Architecture Software Developer's Manual*, as relevant to particular features of the architecture. This chapter provides a collection of all the relevant compatibility information for all Intel Architecture processors and also describes the basic differences with respect to the 16-bit Intel Architecture processors (the Intel 8086 and Intel 286 processors).

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

Appendix B — Model Specific Registers (MSRs). Lists the MSRs available in the Pentium Pro processor and their functions.

1.4. NOTATIONAL CONVENTIONS

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

1.4.1. 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. Intel Architecture 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.

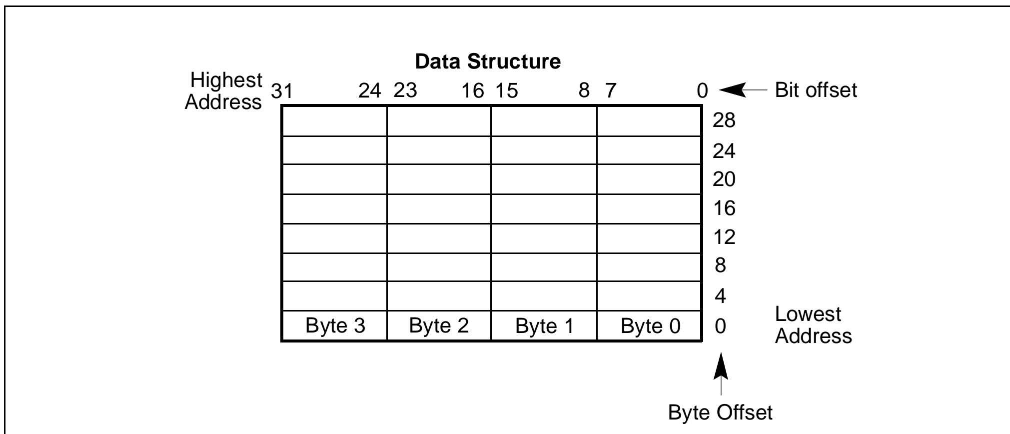


Figure 1-1. Bit and Byte Order

1.4.2. 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 Intel Architecture 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.

1.4.3. Instruction Operands

When instructions are represented symbolically, a subset of the Intel Architecture 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.

1.4.4. 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.

1.4.5. 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 memory. The range of memory that can be addressed is called an **address space**.

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

Segment-register:Byte-address

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

```
DS:FF79H
```

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

```
CS:EIP
```

1.4.6. Exceptions

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

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

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

```
#GP(0)
```

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

1.5. RELATED LITERATURE

The following books contain additional material related to Intel processors:

- *Intel Pentium® Pro Processor Specification Update*, Order Number 242689.

- *Intel Pentium® Processor Specification Update*, Order Number 242480.
- AP-485, *Intel Processor Identification and the CPUID Instruction*, Order Number 241618.
- AP-578, *Software and Hardware Considerations for FPU Exception Handlers for Intel Architecture Processors*, Order Number 242415-001.
- *Pentium® Pro Processor Family Developer's Manual, Volume 1: Specifications*, Order Number 242690-001.
- *Pentium® Processor Family Developer's Manual,* Order Number 241428.
- *Intel486™ Microprocessor Data Book*, Order Number 240440.
- *Intel486™ SX CPU/Intel487™ SX Math Coprocessor Data Book*, Order Number 240950.
- *Intel486™ DX2 Microprocessor Data Book*, Order Number 241245.
- *Intel486™ Microprocessor Product Brief Book*, Order Number 240459.
- *Intel386™ Processor Hardware Reference Manual*, Order Number 231732.
- *Intel386™ Processor System Software Writer's Guide*, Order Number 231499.
- *Intel386™ High-Performance 32-Bit CHMOS Microprocessor with Integrated Memory Management*, Order Number 231630.
- *376 Embedded Processor Programmer's Reference Manual*, Order Number 240314.
- *80387 DX User's Manual Programmer's Reference*, Order Number 231917.
- *376 High-Performance 32-Bit Embedded Processor*, Order Number 240182.
- *Intel386™ SX Microprocessor*, Order Number 240187.
- *Microprocessor and Peripheral Handbook* (Vol. 1), Order Number 230843.
- *Intel Architecture Optimization Manual*, Order Number 242816.

intel®

2

Introduction to the Intel Architecture

CHAPTER 2
INTRODUCTION TO THE INTEL ARCHITECTURE

A strong case can be made that the exponential growth of both the power and breadth of usage of the computer has made it the most important force that is reshaping human technology, business, and society in the second half of the twentieth century. Further, the computer promises to continue to dominate technological growth well into the twenty-first century, in part since other powerful technological forces that are just emerging are strongly dependent on the growth of computing power for their own existence and growth (such as the Internet, and genetics developments like recombinant DNA research and development). The Intel Architecture is clearly today's preferred computer architecture, as measured by number of computers in use and total computing power available in the world. Thus it is hard to overestimate the importance of the Intel Architecture.

2.1. BRIEF HISTORY OF THE INTEL ARCHITECTURE

The developments leading to the Intel Architecture can be traced back through the 8085 and 8080 microprocessors to the 4004 microprocessor (the first microprocessor, designed by Intel in 1969). However, the first actual processor in the Intel Architecture family is the 8086, quickly followed by a more cost effective version for smaller systems, the 8088. The object code programs created for these processors starting in 1978 will still execute on the latest members of the Intel Architecture family.

The 8086 has 16 bit registers and a 16 bit external data bus, with 20 bit addressing giving a 1-MByte address space. The 8088 is identical except for a smaller external data bus of 8 bits. These processors introduced Intel Architecture segmentation, but only in "Real Mode": 16-bit registers can act as pointers to address into segments of up to 64 KBytes in size. The four segment registers hold the (effectively) 20-bit base addresses of the currently active segments; up to 256 KBytes can be addressed without switching between segments, and a total address range of 1 MByte is available.

The Intel 80286 processor introduced the Protected Mode into the Intel Architecture. This new mode uses the segment register contents as selectors or pointers into descriptor tables. The descriptors provide 24-bit base addresses, allowing a maximum physical memory size of up to 16 MBytes, support for virtual memory management on a segment swapping basis, and various protection mechanisms. These include segment limit checking, read only and execute only segment options, and up to four privilege levels to protect operating system code (in several subdivisions, if desired) from application or user programs. Furthermore, hardware task switching and the Local Descriptor Tables allow the operating system to protect application or user programs from each other.

The Intel386 processor introduced 32-bit registers into the architecture, for use both as operands for calculations and for addressing. The lower half of each 32-bit register retained the properties of one of the 16-bit registers of the earlier two generations, to provide complete upward compat-

ibility. A new virtual-8086 mode was provided to yield greater efficiency when executing programs created for the 8086 and 8088 processors on the new 32-bit machine. The 32-bit addressing was supported with an external 32-bit address bus, giving a 4-GByte address space, and also allowed each segment to be as large as 4 GBytes. The original instructions were enhanced with new 32-bit operand and addressing forms, and completely new instructions were provided, including those for bit manipulation. The Intel386 processor also introduced paging into the Intel Architecture, with the fixed 4-KByte page size providing a method for virtual memory management that was significantly superior compared to using segments for the purpose (it was much more efficient for operating systems, and completely transparent to the applications without significant sacrifice of execution speed). Furthermore, the ability to define segments as large as the 4 GBytes physical address space, together with paging, allowed the creation of protected “flat model”[1] addressing systems in the architecture, including complete implementations of the widely used main-frame operating system UNIX.

The Intel Architecture has been and is committed to the task of maintaining backward compatibility at the object code level to preserve our customers’ very large investment in software, but at the same time, in each generation of the architecture the latest most effective microprocessor architecture and silicon fabrication technologies have been used to produce the fastest, most powerful processors possible. Intel has worked over the generations to adapt and incorporate increasingly sophisticated techniques from main-frame architecture into microprocessor architecture. Various forms of parallel processing have been the most performance enhancing of these techniques, and the Intel386 processor was the first Intel Architecture processor to include a number of parallel stages: six. These are the Bus Interface Unit (accesses memory and I/O for the other units), the Code Prefetch Unit (receives object code from the Bus Unit and puts it into a 16 byte queue), the Instruction Decode Unit (decodes object code from the Prefetch unit into microcode), the Execution Unit (executes the microcode instructions), the Segment Unit (translates logical addresses to linear addresses and does protection checks), and the Paging Unit (translates linear addresses to physical addresses, does page based protection checks, and contains a cache with information for up to 32 most recently accessed pages).

The Intel486 processor added more parallel execution capability by (basically) expanding the Intel386 processor’s Instruction Decode and Execution Units into five pipelined stages, where each stage (when needed) operates in parallel with the others on up to five instructions in different stages of execution. Each stage can do its work on one instruction in one clock, and so the Intel486 processor can execute as rapidly as one instruction per CPU clock. An 8-KByte on chip L1 cache was added to the Intel486 processor to greatly increase the percent of instructions that could execute at the scalar rate of one per clock: memory access instructions were now included if the operand was in the L1 cache. The Intel486 processor also for the first time integrated the Floating-Point math Unit onto the same chip as the CPU (see section 2.3 below) and added new pins, bits and instructions to support more complex and powerful systems (L2 cache support and multiprocessor support).

Late in the Intel486 processor generation, Intel incorporated features designed to support energy savings and other system management capabilities into the Intel Architecture mainstream with the Intel486 SL Enhanced processors. These features were developed in the Intel386 SL and Intel486 SL processors, which were specialized for the rapidly growing battery operated note-

1. Requires ony one 32-bit address component to access anywhere in the address space.

book PC market. The features include the new System Management Mode, triggered by its own dedicated interrupt pin, which allows complex system management features (such as power management of various subsystems within the PC), to be added to a system transparently to the main operating system and all applications. The Stop Clock and Auto Halt Powerdown features allow the CPU itself to execute at a reduced clock rate to save power, or to be shut down (with state preserved) to save even more power.

The Intel Pentium processor added a second execution pipeline to achieve superscalar performance (two pipelines, known as u and v, together can execute two instructions per clock). The on-chip L1 cache has also been doubled, with 8 KBytes devoted to code, and another 8 KBytes to data. The data cache uses the MESI protocol to support the more efficient write-back mode, as well as the write-through mode that is used by the Intel486 processor. Branch prediction with an on-chip branch table has been added to increase performance in looping constructs. Extensions have been added to make the virtual-8086 mode more efficient, and to allow for 4-MByte as well as 4-KByte pages. The main registers are still 32 bits, but internal data paths of 128 and 256-bits have been added to speed internal data transfers, and the burstable external data bus has been increased to 64 bits. The Advanced Programmable Interrupt Controller (APIC) has been added to support systems with multiple Pentium processors, and new pins and a special mode (dual processing) has been designed in to support glueless two processor systems.

The Intel Pentium Pro processor is the latest and most powerful member of the Intel Architecture. It has three-way superscalar architecture, which means that it can execute three instructions per CPU clock. It does this by incorporating even more parallelism than the Pentium processor. The Pentium Pro processor provides Dynamic Execution (micro-data flow analysis, out-of-order execution, superior branch prediction, and speculative execution) in a superscalar implementation. Three instruction decode units work in parallel to decode object code into smaller operations called “micro-ops.” These go into an instruction pool, and (when interdependencies don’t prevent) can be executed out of order by the five parallel execution units (two integer, two FPU and one memory interface unit). The Retirement Unit retires completed micro-ops in their original program order, taking account of any branches. The power of the Pentium Pro processor is further enhanced by its caches: it has the same two on-chip 8-KByte L1 caches as does the Pentium processor, and also has a 256-KByte L2 cache that’s in the same package as, and closely coupled to, the CPU, using a dedicated 64-bit (“backside”) full clock speed bus. The L1 cache is dual ported, the L2 cache supports up to 4 concurrent accesses, and the 64-bit external data bus is transaction-oriented, meaning that each access is handled as a separate request and response, with numerous requests allowed while awaiting a response. These parallel features for data access work with the parallel execution capabilities to provide a “non-blocking” architecture in which the processor is more fully utilized and performance is enhanced. The Pentium Pro processor also has an expanded 36-bit address bus, giving a maximum physical address space of 64 GBytes.

Since the Pentium Pro processor is currently the most advanced of the Intel Architecture family, a more detailed description of its architecture is provided in Sections 2.4. and 2.5. More detailed hardware and architectural information on each of the generations of the Intel Architecture family is available in the separate data books for the processor generations (see Section 1.5., *Related Literature*).

2.2. INCREASING INTEL ARCHITECTURE PERFORMANCE AND MOORE'S LAW

In the mid-1960s, Intel Chairman of the Board Gordon Moore deduced a principle or "law" which has continued to be true for over three decades: the computing power and the complexity (or roughly, the number of transistors per CPU chip) of the silicon integrated circuit microprocessor doubles every one to two years, and the cost per CPU chip is cut in half. This law is the main explanation for the computer revolution, in which the Intel Architecture plays such a significant role.

The table below shows the dramatic increases in performance and transistor count of the Intel Architecture processors over their history, as predicted by Moore's Law, and also summarizes the evolution of other key features of the architecture.

Table 2-1. Processor Performance Over Time and Other Key Features of the Intel Architecture

Intel Processor	Date of Product Intro-duction	Perfor-mance in MIPs[1]	Max. CPU Frequency at Intro-duction	No. of Transis-tors on the Die	Main CPU Register Size[2]	Extern. Data Bus Size[2]	Max. Extern. Addr. Space	Caches in CPU Pack-age[3]
8086	1978	0.8	8 MHz	29 K	16	16	1 MB	None
Intel 286	1982	2.7	12.5 MHz	134 K	16	16	16 MB	Note 3
Intel386™ DX	1985	6.0	20 MHz	275 K	32	32	4 GB	Note 3
Intel486™ DX	1989	20	25 MHz	1.2 M	32	32	4 GB	8KB L1
Pentium®	1993	100	60 MHz	3.1 M	32	64	4 GB	16KB L1
Pentium Pro	1995	440	200 MHz	5.5 M	32	64	64 GB	16KB L1; 256KB or 512KB L2

NOTES:

1. Performance here is indicated by Dhrystone MIPs (Millions of Instructions per Second) because even though MIPs are no longer considered a preferred measure of CPU performance, they are the only benchmarks that span all six generations of the Intel Architecture. The MIPs and frequency values given here correspond to the maximum CPU frequency available at product introduction.
2. Main CPU register size and external data bus size are given in bits. Note also that there are 8 and 16-bit data registers in all of the CPUs, there are eight 80-bit registers in the FPUs integrated into the Intel386™ chip and beyond, and there are internal data paths that are 2 to 4 times wider than the external data bus for each processor.
3. In addition to the large general purpose caches listed in the table for the Intel486™ processor (8 KBytes of combined code and data) and the Intel Pentium® and Pentium Pro processors (8 KBytes each for separate code cache and data cache), there are smaller special purpose caches. The Intel 286 has 6 byte descriptor caches for each segment register. The Intel386 has 8 byte descriptor caches for each segment register, and also a 32 entry, 4 way set associative Translation Lookaside Buffer (cache) to store access information for recently used pages on the chip. The Intel486 has the same caches described for the Intel386, as well as its 8K L1 general purpose cache. The Intel Pentium and Pentium Pro processors have their general purpose caches, descriptor caches, and two Translation Lookaside Buffers each (one for each 8K L1 cache).

2.3. BRIEF HISTORY OF THE INTEL ARCHITECTURE FLOATING-POINT UNIT

The Intel Architecture Floating-Point Units (FPUs) before the Intel486 lack the added efficiency of integration into the CPU, but have provided the option of greatly enhanced floating-point performance since the beginning of the family. (Since the earlier FPUs were on separate chips, they were often referred to as numeric processor extensions (NPXs) or math coprocessors (MCPs).) With each succeeding generation, Intel has made significant increases in the power and flexibility of the FPU, and yet has maintained complete upward compatibility. The Pentium Pro Processor offers compatibility with object code for 8087, Intel 287, Intel 387 DX, Intel 387 SX, and Intel 487 DX math coprocessors and the Intel486 DX and Pentium processors.

The 8087 numeric processor extension (NPX) was designed for use in 8086-family systems. The 8086 was the first microprocessor family to partition the processing unit to permit high-performance numeric capabilities. The 8087 NPX for this processor family implemented a complete numeric processing environment in compliance with an early proposal for IEEE Standard 754 for Binary Floating-Point Arithmetic.

With the Intel 287 coprocessor NPX, high-speed numeric computations were extended to 80286 high-performance multitasking and multi-user systems. Multiple tasks using the numeric processor extension were afforded the full protection of the 80286 memory management and protection features.

The Intel 387 DX and SX math coprocessors are Intel's third generation numeric processors. They implement the final IEEE Std 754, adding new trigonometric instructions, and using a new design and CHMOS-III process to allow higher clock rates and require fewer clocks per instruction. Together, the Intel 387 math coprocessor with additional instructions and the improved standard brought even more convenience and reliability to numeric programming and made this convenience and reliability available to applications that need the high-speed and large memory capacity of the 32-bit environment of the Intel386 microprocessor.

The Intel486 processor FPU is an on-chip equivalent of the Intel 387 DX math coprocessor conforming to both IEEE Std 754 and the more recent, generalized IEEE Std 854. Having the FPU on chip results in a considerable performance improvement in numeric-intensive computation.

The Pentium processor FPU has been completely redesigned over the Intel486 processor FPU while maintaining conformance to both the IEEE Std 754 and 854. Faster algorithms provide at least three times the performance over the Intel486 processor FPU for common operations including ADD, MUL, and LOAD. Many applications can achieve five times the performance of the Intel486 processor FPU or more with instruction scheduling and pipelined execution.

2.4. INTRODUCTION TO THE PENTIUM® PRO PROCESSOR'S ADVANCED MICROARCHITECTURE

The Pentium Pro processor (introduced by Intel in 1995) represents the most recent implementation of the Intel Architecture. Like its predecessor, the Pentium processor (introduced by Intel in 1993), the Pentium Pro processor, with its advanced superscalar microarchitecture, sets an impressive performance standard. In designing the Pentium Pro processor, one of the primary

goals of the Intel chip architects was to exceed the performance of the Pentium processor significantly while still using the same 0.6-micrometer, four-layer, metal BICMOS manufacturing process. Using the same manufacturing process as the Pentium processor meant that performance gains could only be achieved through substantial advances in the microarchitecture.

The resulting Pentium Pro processor microarchitecture is a three-way superscalar, pipelined architecture. The term "three-way superscalar" means that using parallel processing techniques, the processor is able on average to decode, dispatch, and complete execution of (retire) three instructions per clock cycle. To handle this level of instruction throughput, the Pentium Pro processor uses a decoupled, 12-stage superpipeline that supports out-of-order instruction execution. Figure 2-1 shows a conceptual view of this pipeline, with the pipeline divided into four processing units (the fetch/decode unit, the dispatch/execute unit, the retire unit, and the instruction pool). Instructions and data are supplied to these units through the bus interface unit.

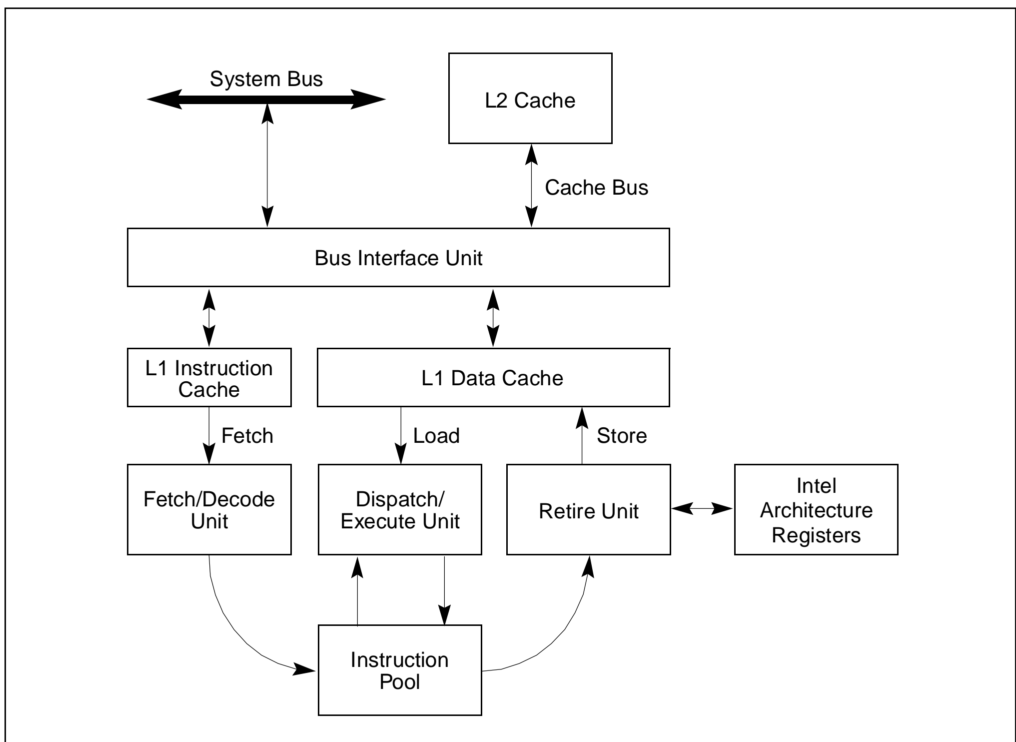


Figure 2-1. The Processing Units in the Pentium® Pro Processor Microarchitecture and Their Interface with the Memory Subsystem

To insure a steady supply of instructions and data to the instruction execution pipeline, the Pentium Pro processor microarchitecture incorporates two cache levels. The L1 cache provides an 8-KByte instruction cache and an 8-KByte data cache, both closely coupled to the pipeline. The L2 cache is a 256-KByte static RAM that is coupled to the core processor through a full clock-speed, 64-bit, cache bus.

The centerpiece of the Pentium Pro processor microarchitecture is an innovative out-of-order execution mechanism called "dynamic execution." Dynamic execution incorporates three data-processing concepts:

- Deep branch prediction.
- Dynamic data flow analysis.
- Speculative execution.

Branch prediction is a concept found in most mainframe and high-speed microprocessor architectures. It allows the processor to decode instructions beyond branches to keep the instruction pipeline full. In the Pentium Pro processor, the instruction fetch/decode unit uses a highly optimized branch prediction algorithm to predict the direction of the instruction stream through multiple levels of branches, procedure calls, and returns.

Dynamic data flow analysis involves real-time analysis of the flow of data through the processor to determine data and register dependencies and to detect opportunities for out-of-order instruction execution. The Pentium Pro processor dispatch/execute unit can simultaneously monitor many instructions and execute these instructions in the order that optimizes the use of the processor's multiple execution units, while maintaining the integrity of the data being operated on. This out-of-order execution keeps the execution units busy even when cache misses and data dependencies among instructions occur.

Speculative execution refers to the processor's ability to execute instructions ahead of the program counter but ultimately to commit the results in the order of the original instruction stream. To make speculative execution possible, the Pentium Pro processor microarchitecture decouples the dispatching and executing of instructions from the commitment of results. The processor's dispatch/execute unit uses data-flow analysis to execute all available instructions in the instruction pool and store the results in temporary registers. The retirement unit then linearly searches the instruction pool for completed instructions that no longer have data dependencies with other instructions or unresolved branch predictions. When completed instructions are found, the retirement unit commits the results of these instructions to memory and/or the Intel Architecture registers (the processor's eight general-purpose registers and eight floating-point unit data registers) in the order they were originally issued and retires the instructions from the instruction pool.

Through deep branch prediction, dynamic data-flow analysis, and speculative execution, dynamic execution removes the constraint of linear instruction sequencing between the traditional fetch and execute phases of instruction execution. It allows instructions to be decoded deep into multi-level branches to keep the instruction pipeline full. It promotes out-of-order instruction execution to keep the processor's six instruction execution units running at full capacity. And finally it commits the results of executed instructions in original program order to maintain data integrity and program coherency.

The following section describes the Pentium Pro processor microarchitecture in greater detail.

2.5. DETAILED DESCRIPTION OF THE PENTIUM® PRO PROCESSOR MICROARCHITECTURE

Figure 2-2 shows a functional block diagram of the Pentium Pro processor microarchitecture. In this diagram, the following blocks make up the four processing units and the memory subsystem shown in Figure 2-1:

- Memory subsystem—System bus, L2 cache, bus interface unit, instruction cache (L1), data cache unit (L1), memory interface unit, and memory reorder buffer.
- Fetch/decode unit—Instruction fetch unit, branch target buffer, instruction decoder, microcode sequencer, and register alias table.
- Instruction pool—Reorder buffer
- Dispatch/execute unit—Reservation station, two integer units, two floating-point units, and two address generation units.
- Retire unit—Retire unit and retirement register file.

2.5.1. Memory Subsystem

The memory subsystem for the Pentium Pro processor consists of main system memory, the primary cache (L1), and the secondary cache (L2). The bus interface unit accesses system memory through the external system bus. This 64-bit bus is a transaction-oriented bus, meaning that each bus access is handled as separate request and response operations. While the bus interface unit is waiting for a response to one bus request, it can issue numerous additional requests.

The bus interface unit accesses the close-coupled L2 cache through a 64-bit cache bus. This bus is also transactional oriented, supporting up to four concurrent cache accesses, and operates at the full clock speed of the processor.

Access to the L1 caches is through internal buses, also at full clock speed. The 8-KByte L1 instruction cache is four-way set associative; the 8-KByte L1 data cache is dual-ported and two-way set associative, supporting one load and one store operation per cycle.

Coherency between the caches and system memory are maintained using the MESI (modified, exclusive, shared, invalid) cache protocol. This protocol fosters cache coherency in single- and multiple-processor systems. It is also able to detect coherency problems created by self-modifying code.

Memory requests from the processor's execution units go through the memory interface unit and the memory order buffer. These units have been designed to support a smooth flow of memory access requests through the cache and system memory hierarchy to prevent memory access blocking. The L1 data cache automatically forwards a cache miss on to the L2 cache, and then, if necessary, the bus interface unit forwards an L2 cache miss to system memory.

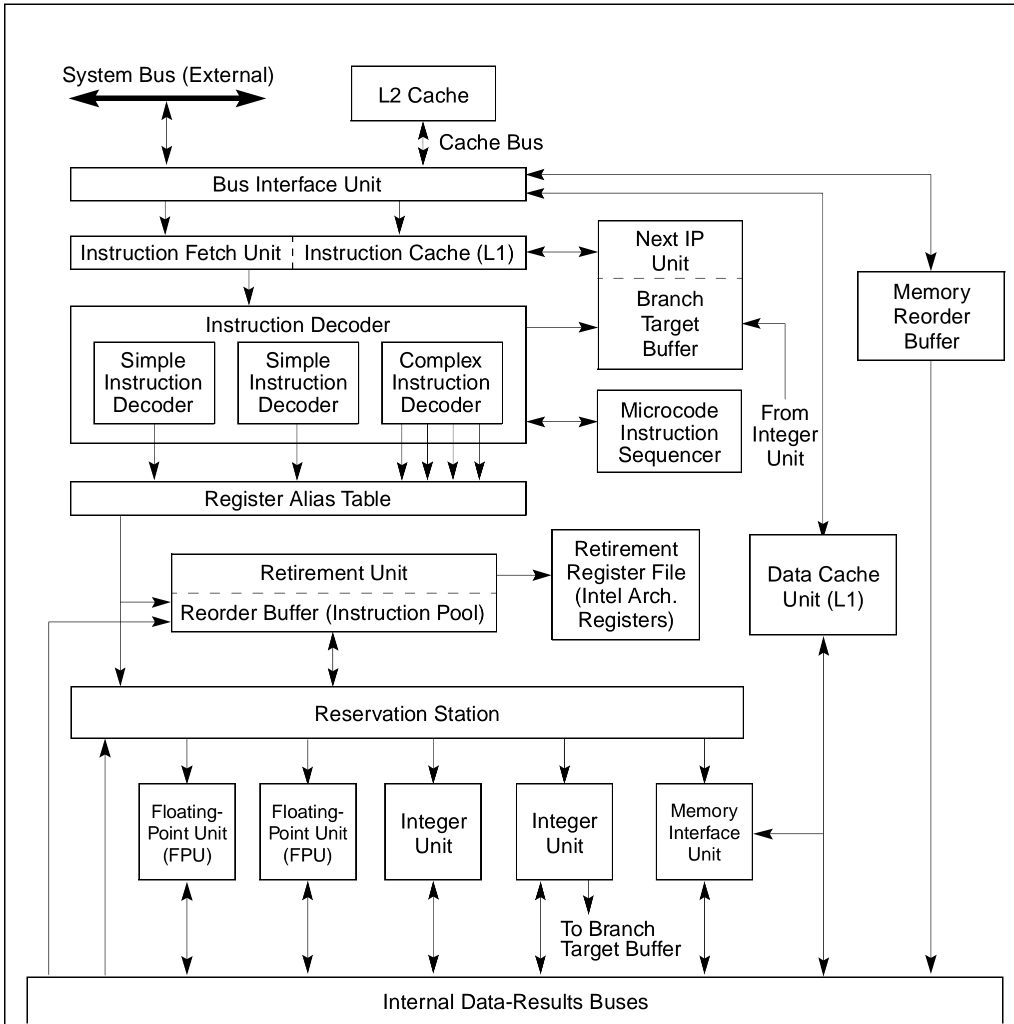


Figure 2-2. Functional Block Diagram of the Pentium® Pro Processor Microarchitecture

Memory requests to the L2 cache or system memory go through the memory reorder buffer, which functions as a scheduling and dispatch station. This unit keeps track of all memory requests and is able to reorder some requests to prevent blocks and improve throughput. For example, the memory reorder buffer allows loads to pass stores. It also issues speculative loads. (Stores are always dispatched in order, and speculative stores are never issued.)

2.5.2. The Fetch/Decode Unit

The fetch/decode unit reads a stream of Intel Architecture instructions from the L1 instruction cache and decodes them into a series of micro-operations called “micro-ops.” This micro-op stream (still in the order of the original instruction stream) is then sent to the instruction pool.

The instruction fetch unit fetches one 32-byte cache line per clock from the instruction cache. It marks the beginning and end of the Intel Architecture instructions in the cache lines and transmits 16 aligned bytes to the decoder.

The instruction fetch unit computes the instruction pointer, based on inputs from the branch target buffer, the exception/interrupt status, and branch-misprediction indications from the integer execution units. The most important part of this process is the branch prediction performed by the branch target buffer. Using an extension of Yeh’s algorithm, the 512 entry branch target buffer looks many instructions ahead of the retirement program counter. Within this instruction window there may be numerous branches, procedure calls, and returns that must be correctly predicted if the dispatch/execute unit is to do useful work.

The instruction decoder contains three parallel decoders: two simple-instruction decoders and one complex instruction decoder. Each decoder converts an Intel Architecture instruction into one or more triadic micro-ops (two logical sources and one logical destination per micro-op). Micro-ops are primitive instructions that are executed by the processor’s six parallel execution units.

Many Intel Architecture instructions are converted directly into single micro-ops by the simple instruction decoders, and some instructions are decoded into from one to four micro-ops. The more complex Intel Architecture instructions are decoded into sequences of preprogrammed micro-ops obtained from the microcode instruction sequencer. The instruction decoders also handle the decoding of instruction prefixes and looping operations. The instruction decoder can generate up to six micro-ops per clock cycle (one each for the simple instruction decoders and four for the complex instruction decoder).

The Intel Architecture’s register set can cause resource stalls due to register dependencies. To solve this problem, the processor provides 40 internal, general-purpose registers, which are used for the actual computations. These registers can handle both integer and floating-point values. To allocate the internal registers, the enqueued micro-ops from the instruction decoder are sent to the register alias table unit, where references to the logical Intel Architecture registers are converted into internal physical register references.

In the final step of the decoding process, the allocator in the register alias table unit adds status bits and flags to the micro-ops to prepare them for out-of-order execution and sends the resulting micro-ops to the instruction pool.

2.5.3. Instruction Pool (Reorder Buffer)

Prior to entering the instruction pool (known formally as the reorder buffer), the micro-op instruction stream is in the same order as the Intel Architecture instruction stream that was sent to the instruction decoder. No reordering of instructions has taken place.

The reorder buffer is an array of content-addressable memory, arranged into 40 micro-op registers. It contains micro-ops that are waiting to be executed, as well as those that have already been executed but not yet committed to machine state. The dispatch/execute unit can execute instructions from the reorder buffer in any order.

2.5.4. Dispatch/Execute Unit

The dispatch/execute unit is an out-of-order unit that schedules and executes the micro-ops stored in the reorder buffer according to data dependencies and resource availability and temporarily stores the results of these speculative executions.

The scheduling and dispatching of micro-ops from the reorder buffer is handled by the reservation station. It continuously scans the reorder buffer for micro-ops that are ready to be executed (that is, all the source operands are available) and dispatches them to the available execution units. The results of a micro-op execution are returned to the reorder buffer and stored along with the micro-op until it is retired. This scheduling and dispatching process supports classic out-of-order execution, where micro-ops are dispatched to the execution units strictly according to data-flow constraints and execution resource availability, without regard to the original ordering of the instructions. When two or more micro-ops of the same type (for example, integer operations) are available at the same time, they are executed in a pseudo FIFO order in the reorder buffer.

Execution of micro-ops is handled by two integer units, two floating-point units, and one memory-interface unit, allowing up to five micro-ops can be scheduled per clock.

The two integer units can handle two integer micro-ops in parallel. One of the integer units is designed to handle branch micro-ops. This unit has the ability to detect branch mispredictions and signal the branch target buffer to restart the pipeline. This operation is handled as follows. The instruction decoder tags each branch micro-op with both branch destination addresses (the predicted destination and the fall-through destination). When the integer unit executes the branch micro-op, it is able to determine whether the predicted or the fall-through destination was taken. If the predicted branch is taken, then speculatively executed micro-ops are marked usable and execution continues along the predicted instruction path. If the predicted branch was not taken, a jump execution unit in the integer unit changes the status of all of the micro-ops following the branch to remove them from the instruction pool. It then provides the proper branch destination to the branch target buffer, which in turn restarts the pipeline from the new target address.

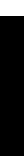
The memory interface unit handles load and store micro-ops. A load access only needs to specify the memory address, so it can be encoded in one micro-op. A store access needs to specify both an address and the data to be written, so it is encoded in two micro-ops. The part of the memory interface unit that handles stores has two ports allowing it to process the address and the data micro-op in parallel. The memory interface unit can thus execute both a load and a store in parallel in one clock cycle.

The floating-point execution units are similar to those found in the Pentium processor. Several new floating-point instructions have been added to the Pentium Pro processor to streamline conditional branches and moves.

2.5.5. Retirement Unit

The retirement unit commits the results of speculatively executed micro-ops to permanent machine state and removes the micro-ops from the reorder buffer. Like the reservation station, the retirement unit continuously checks the status of micro-ops in the reorder buffer, looking for ones that have been executed and no longer have any dependencies with other micro-ops in the instruction pool. It then retires completed micro-ops in their original program order, taking into accounts interrupts, exceptions, breakpoints, and branch mispredictions.

The retirement unit can retire three micro-ops per clock. In retiring a micro-op, it writes the results to the retirement register file and/or memory. The retirement register file contains the Intel Architecture registers (eight general-purpose registers and eight floating-point data registers). After the results have been committed to machine state, the micro-op is removed from the reorder buffer.



3

Basic Execution Environment

CHAPTER 3 BASIC EXECUTION ENVIRONMENT

This chapter describes the basic execution environment of an Intel Architecture processor as seen by assembly-language programmers. It describes how the processor executes instructions and how it stores and manipulates data. The parts of the execution environment described here include memory (the address space), the general-purpose data registers, the segment registers, the EFLAGS register, and the instruction pointer register.

The execution environment for the floating-point unit (FPU) is described in Chapter 7, *Floating-Point Unit*.

3.1. MODES OF OPERATION

The Intel Architecture supports three operating modes: protected mode, real-address mode, and system management mode. The operating mode determines which instructions and architectural features are accessible:

- **Protected mode.** The native state of the processor. In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode for all new applications and operating systems.

  Among the capabilities of protected mode is the ability to directly execute "real-address mode" 8086 software in a protected, multi-tasking environment. This feature is called **virtual-8086 mode**, although it is not actually a processor mode. Virtual-8086 mode is actually a protected mode attribute that can be enabled for any task.

- **Real-address mode.** Provides the programming environment of the Intel 8086 processor with a few extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.

- **System management mode.** A standard architectural feature unique to all Intel processors, beginning with the Intel386 SL processor. This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC). In SMM, the processor switches to a separate address space while saving the entire context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt.

The basic execution environment is the same for each of these operating modes, as is described in the remaining sections of this chapter.

3.2. OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT

Any program or task running on an Intel Architecture processor is given a set of resources for executing instructions and for storing code, data, and state information. These resources (shown in Figure 3-1) include an address space of up to 2^{32} bytes, a set of general data registers, a set of segment registers, and a set of status and control registers. When a program calls a procedure, a procedure stack is added to the execution environment. (Procedure calls and the procedure stack implementation are described in Chapter 4, *Procedure Calls, Interrupts, and Exceptions*.)

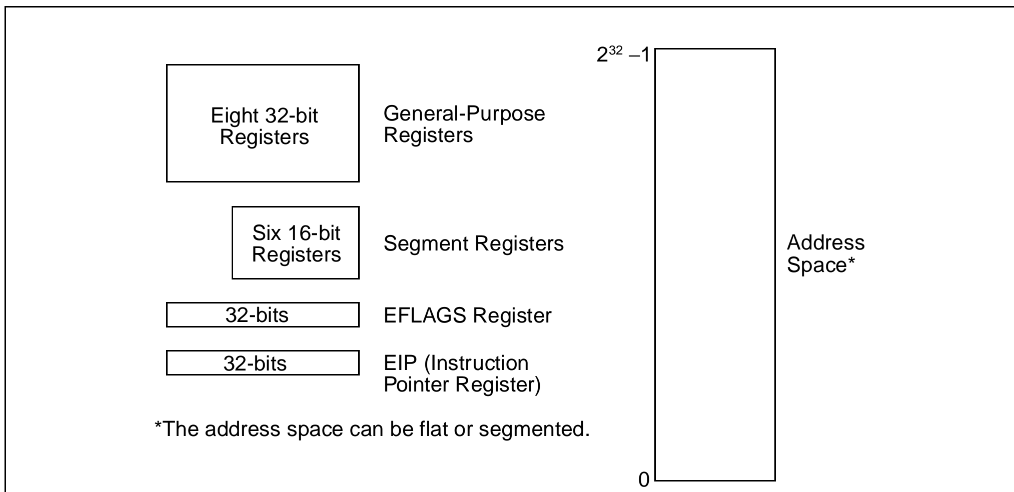


Figure 3-1. Pentium® Pro Processor Basic Execution Environment

3.3. MEMORY ORGANIZATION

The memory that the processor addresses on its bus is called **physical memory**. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a **physical address**. The **physical address space** ranges from zero to a maximum of $2^{32}-1$ (4 gigabytes).

Virtually any operating system or executive designed to work with an Intel Architecture processor will use the processor's memory management facilities to access memory. These facilities provide features such as segmentation and paging, which allow memory to be managed efficiently and reliably. Memory management is described in detail in Chapter 3, *Protected-Mode Memory Management*, of the *Intel Architecture Software Developer's Manual, Volume 3*. The following paragraphs describe the basic methods of addressing memory when memory management is used.

When employing the processor's memory management facilities, programs do not directly address physical memory. Instead, they access memory using any of three memory models: flat, segmented, or real-address mode.

With the **flat** memory model (see Figure 3-2), memory appears to a program as a single, continuous address space, called a **linear address space**. Code (a program's instructions), data, and the procedure stack are all contained in this address space. The linear address space is byte addressable, with addresses running contiguously from 0 to $2^{32} - 1$. An address for any byte in the linear address space is called a **linear address**.

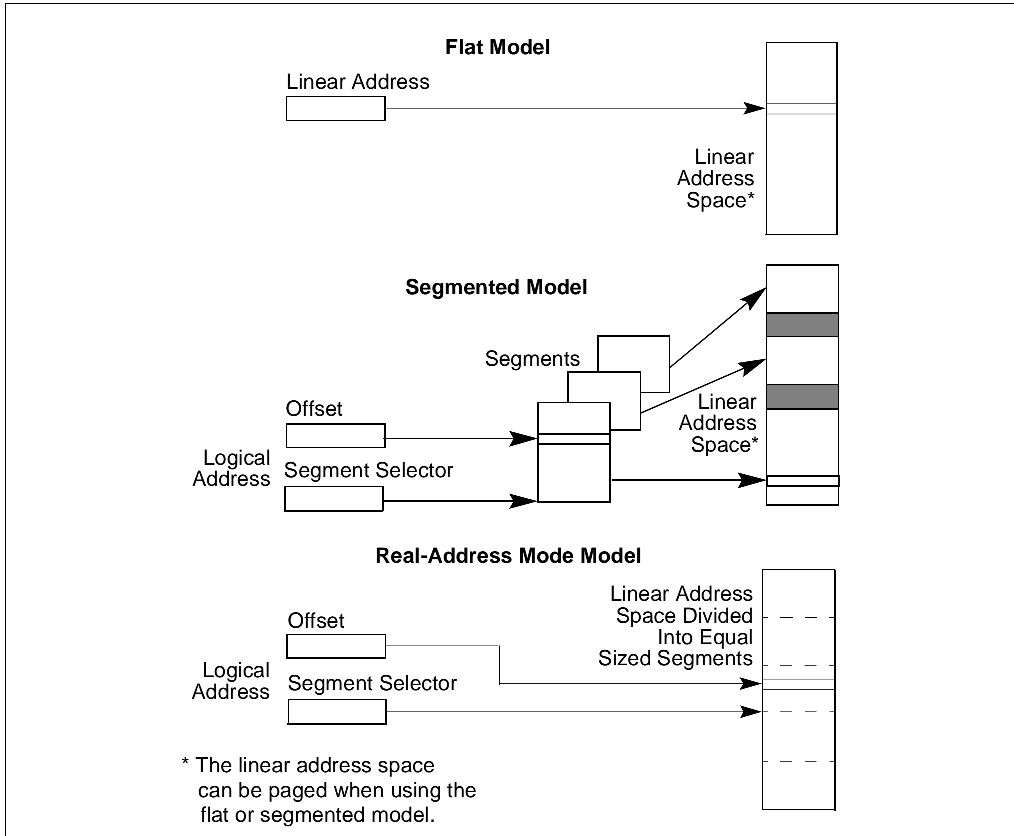


Figure 3-2. Three Memory Management Models

With the **segmented** memory model, memory appears to a program as a group of independent address spaces called **segments**. When using this model, code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program must issue a **logical address**, which consists of a segment selector and an offset. (A logical address is often referred to as a **far pointer**.) The **segment selector** identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. The programs running on an Intel Architecture processor can address up to 16,383 segments of different sizes and types, and each segment can be as large as 2^{32} bytes.

Internally, all the segments that are defined for a system are mapped into the processor's linear address space. So, the processor translates each logical address into a linear address to access a memory location. This translation is transparent to the application program.

The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program's stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively. And placing the operating system's or executive's code, data, and stack in separate segments protects them from the application program and vice versa.

With either the flat or segmented model, the Intel Architecture provides facilities for dividing the linear address space into pages and mapping the pages into virtual memory. If an operating system/executive uses the Intel Architecture's paging mechanism, the existence of the pages is transparent to an application program.

The **real-address mode** model uses the memory model for the Intel 8086 processor, the first Intel Architecture processor. It was provided in all the subsequent Intel Architecture processors for compatibility with existing programs written to run on the Intel 8086 processor. The real-address mode uses a specific implementation of segmented memory in which the linear address space for the program and the operating system/executive consists of an array of segments of up to 64K bytes in size each. The maximum size of the linear address space in real-address mode is 2^{20} bytes. (See Chapter 15, *8086 Emulation*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on this memory model.)

3.4. MODES OF OPERATION

When writing code for the Pentium Pro processor, a programmer needs to know the operating mode the processor is going to be in when executing the code and the memory model being used. The relationship between operating modes and memory models is as follows:

- **Protected mode.** When in protected mode, the processor can use any of the memory models described in this section. (The real-addressing mode memory model is ordinarily used only when the processor is in the virtual-8086 mode.) The memory model used depends on the design of the operating system or executive. When multitasking is implemented, individual tasks can use different memory models.
- **Real-address mode.** When in real-address mode, the processor only supports the real-address mode memory model.
- **System management mode.** When in SMM, the processor switches to a separate address space, called the system management RAM (SMRAM). The memory model used to address bytes in this address space is similar to the real-address mode model. (See Chapter 11, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on the memory model used in SMM.)

3.5. 32-BIT VS. 16-BIT ADDRESS AND OPERAND SIZES

The processor can be configured for 32-bit or 16-bit address and operand sizes. With 32-bit address and operand sizes, the maximum linear address or segment offset is FFFFFFFFH (2^{32}), and operand sizes are typically 8 bits or 32 bits. With 16-bit address and operand sizes, the maximum linear address or segment offset is FFFFH (2^{16}), and operand sizes are typically 8 bits or 16 bits.

When using 32-bit addressing, a logical address (or far pointer) consists of a 16-bit segment selector and a 32-bit offset; when using 16-bit addressing, it consists of a 16-bit segment selector and a 16-bit offset.

Instruction prefixes allow temporary overrides of the default address and/or operand sizes from within a program.

When operating in protected mode, the segment descriptor for the currently executing code segment defines the default address and operand size. A segment descriptor is a system data structure not normally visible to application code. Assembler directives allow the default addressing and operand size to be chosen for a program. The assembler and other tools then set up the segment descriptor for the code segment appropriately.

When operating in real-address mode, the default addressing and operand size is 16 bits. An address-size override can be used in real-address mode to enable 32 bit addressing; however, the maximum allowable 32-bit address is still 0000FFFFH (2^{16}).

3.6. REGISTERS

The processor provides 16 registers for use in general system and application programing. As shown in Figure 3-3, these registers can be grouped as follows:

- **General-purpose data registers**. These eight registers are available for storing operands and pointers.
- **Segment registers**. These registers hold up to six segment selectors.
- **Status and control registers**. These registers report and allow modification of the state of the processor and of the program being executed.

3.6.1. General-Purpose Data Registers

The 32-bit general-purpose data registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers.

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for any other purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

The special uses of general-purpose registers by instructions are described in Chapter 6, *Instruction Set Summary*, in this volume and Chapter 3, *Instruction Set Reference*, in the *Intel Architecture Software Developer's Manual, Volume 2*. The following is a summary of these special uses:

- EAX—Accumulator for operands and results data.
- EBX—Pointer to data in the DS segment.

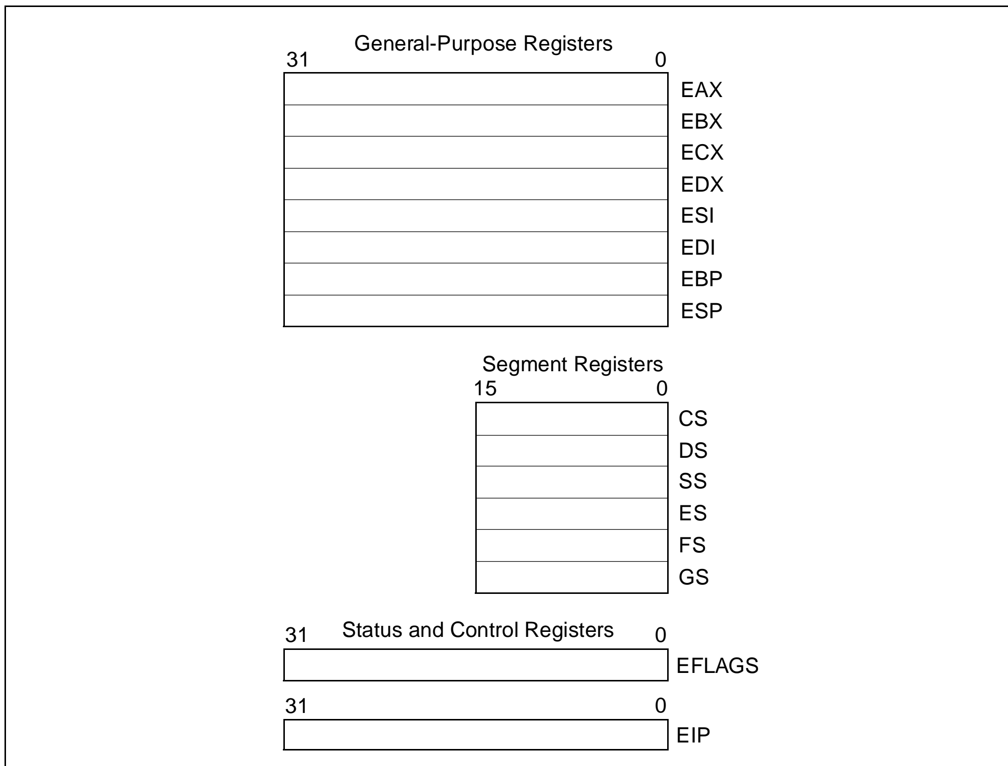


Figure 3-3. Application Programming Registers

- ECX—Counter for string and loop operations.
- EDX—I/O pointer.
- ESI—Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
- EDI—Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
- ESP—Stack pointer (in the SS segment).
- EBP—Pointer to data on the stack (in the SS segment).

As shown in Figure 3-4, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SP, SI, and DI. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

General-Purpose Registers

31–16	15–8	7–0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Figure 3-4. Alternate General-Purpose Register Names

3.6.2. Segment Registers

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

When writing application code, you generally create segment selectors with assembler directives and symbols. The assembler and other tools then create the actual segment selector values associated with these directives and symbols. If you are writing system code, you may need to create segment selectors directly. (A detailed description of the segment-selector data structure is given in Chapter 3, *Protected-Mode Memory Management*, of the *Intel Architecture Software Developer's Manual, Volume 3*.)

How segment registers are used depends on the type of memory management model that the operating system or executive is using. When using the flat (unsegmented) memory model, the segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space (as shown in Figure 3-5). These overlapping segments then comprise the linear-address space for the program. (Typically, two overlapping segments are defined: one for code and another for data and stacks. The CS segment register points to the code segment and all the other segment registers point to the data and stack segment.)

When using the segmented memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear-address space (as shown in Figure 3-6). At any time, a program can thus access up to six segments in the linear-address space. To access a segment not pointed to by one of the segment registers, a program must first load the segment selector for the segment to be accessed into a segment register.

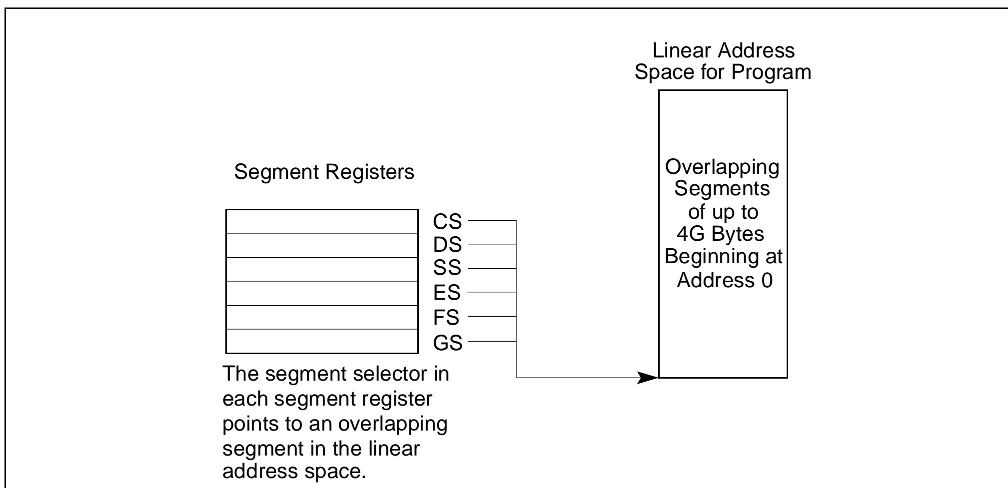


Figure 3-5. Use of Segment Registers for Flat Memory Model

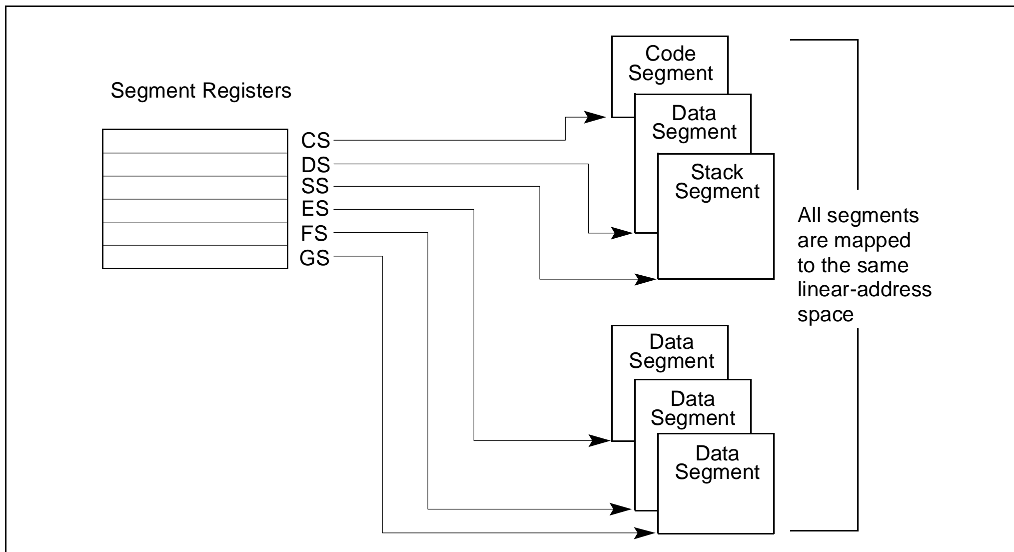


Figure 3-6. Use of Segment Registers in Segmented Memory Model

Each of the segment registers is associated with one of three types of storage: code, data, or stack). For example, the CS register contains the segment selector for the **code segment**, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the linear address within the code segment of the next instruction to be executed. The CS register cannot be loaded explicitly by an application program. Instead, it is loaded implicitly by instructions or internal processor operations that change program control (such as, procedure calls, interrupt handling, or task switching).

The DS, ES, FS, and GS registers point to four **data segments**. The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another program. To access additional data segments, the application program must load segment selectors for these segments into the DS, ES, FS, and GS registers, as needed.

The SS register contains the segment selector for a **stack segment**, where the procedure stack is stored for the program, task, or handler currently being executed. All stack operations use the SS register to find the stack segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up multiple stacks and switch among them.

See Section 3.3., "Memory Organization", for an overview of how the segment registers are used in real-address mode.

The four segment registers CS, DS, SS, and ES are the same as the segment registers found in the Intel 8086 and Intel 286 processors and the FS and GS registers were introduced into the Intel Architecture with the Intel386 family of processors.

3.6.3. EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Figure 3-7 defines the flags within this register. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.

Some of the flags in the EFLAGS register can be modified directly, using special-purpose instructions (described in the following sections). There are no instructions that allow the whole register to be examined or modified directly. However, the following instructions can be used to move groups of flags to and from the procedure stack or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents of the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor’s bit manipulation instructions (BT, BTS, BTR, and BTC).

When suspending a task (using the processor’s multitasking facilities), the processor automatically saves the state of the EFLAGS register in the task state segment (TSS) for the task being suspended. When binding itself to a new task, the processor loads the EFLAGS register with data from the new task’s TSS.

When a call is made to an interrupt or exception handler procedure, the processor automatically saves the state of the EFLAGS registers on the procedure stack. When an interrupt or exception is handled with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.

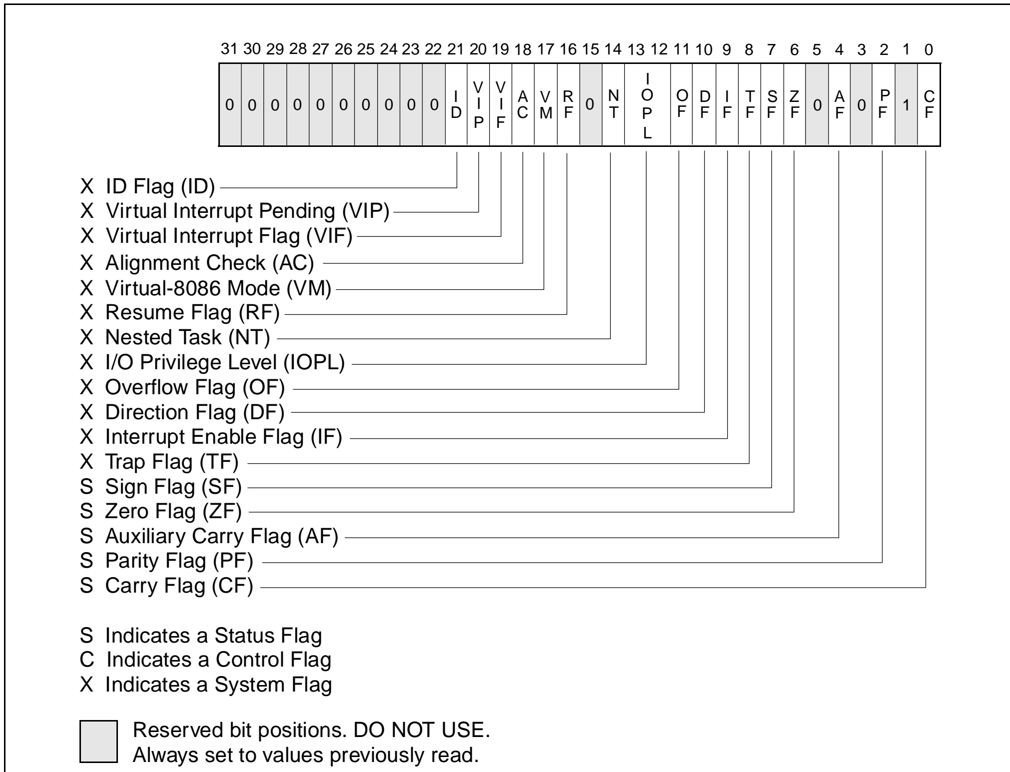


Figure 3-7. EFLAGS Register

As the Intel Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the Intel Architecture processors to the next. As a result, code that accesses or modifies these flags for one family of Intel Architecture processors works as expected when run on later families of processors.

3.6.3.1. STATUS FLAGS

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The functions of the status flags are as follows:

CF (bit 0) **Carry flag.** Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

PF (bit 2) **Parity flag.** Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

AF (bit 4) **Adjust flag.** Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

ZF (bit 6) **Zero flag.** Set if the result is zero; cleared otherwise.

SF (bit 7) **Sign flag.** Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

OF (bit 11) **Overflow flag.** Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer, the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two's complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsigned-integer zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions J*cc* (jump on condition code *cc*), SET*cc* (byte set on condition code *cc*), LOOP*cc*, and CMOV*cc* (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

3.6.3.2. DF FLAG

The direction flag (DF, located in bit 10 of the EFLAGS register) controls the string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (that is, to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).

The STD and CLD instructions set and clear the DF flag, respectively.

3.6.4. System Flags and IOPL Field

The system flags and IOPL field in the EFLAGS register control operating-system or executive operations. **They should not be modified by application programs.** The functions of the status flags are as follows:

IF (bit 9) — **Interrupt enable flag.** Controls the response of the processor to maskable interrupt requests. Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.

TF (bit 8) — **Trap flag.** Set to enable single-step mode for debugging; clear to disable single-step mode.

IOPL (bits 12 and 13) — **I/O privilege level field.** Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. This field can only be modified by the POPF and IRET instructions when operating at a CPL of 0.

NT (bit 14) — **Nested task flag.** Controls the chaining of interrupted and called tasks. Set when the current task is linked to the previously executed task; cleared when the current task is not linked to another task.

RF (bit 16) — **Resume flag.** Controls the processor's response to debug exceptions.

VM (bit 17) — **Virtual-8086 mode flag.** Set to enable virtual-8086 mode; clear to return to protected mode.

AC (bit 18) — **Alignment check flag.** Set this flag and the AM bit in the CR0 register to enable alignment checking of memory references; clear the AC flag and/or the AM bit to disable alignment checking.

VIF (bit 19) — **Virtual interrupt flag.** Virtual image of the IF flag. Used in conjunction with the VIP flag. (To use this flag and the VIP flag the virtual mode extensions are enabled by setting the VME flag in control register CR4.)

VIP (bit 20) — **Virtual interrupt pending flag.** Set to indicate to that an interrupt is pending; clear when no interrupts are pending. (Software sets and clears this flag. The processor only reads it.) Used in conjunction with the VIF flag.

ID (bit 21) — **Identification flag.** The ability of a program to set or clear this flag indicates support for the CPUID instruction.

See Chapter 3, *Protected-Mode Memory Management*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for a detail description of these flags.

3.7. INSTRUCTION POINTER

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, J*cc*, CALL, RET, and IRET instructions.

The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, J*cc*, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET). See Section 4.2.4.2., "Return Instruction Pointer".

All Intel Architecture processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register. Even though different processor generations use different prefetching mechanisms, the function of EIP register to direct program flow remains fully compatible with all software written to run on Intel Architecture processors.

3.8. OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES

When processor is executing in protected mode, every code segment has a default operand-size attribute and address-size attribute. These attributes are selected with the D (default size) flag in the segment descriptor for the code segment (see Chapter 3, *Protected-Mode Memory Management*, in the *Intel Architecture Software Developer's Manual, Volume 3*). When the D flag is set, the 32-bit operand-size and address-size attributes are selected; when the flag is clear, the 16-bit size attributes are selected. When the processor is executing in real-address mode, virtual-8086 mode, or SMM, the default operand-size and address-size attributes are always 16 bits.

The operand-size attribute selects the sizes of operands that instructions operate on. When the 16-bit operand-size attribute is in force, operands can generally be either 8 bits or 16 bits, and when the 32-bit operand-size attribute is in force, operands can generally be 8 bits or 32 bits.

The address-size attribute selects the sizes of addresses used to address memory: 16 bits or 32 bits. When the 16-bit address-size attribute is in force, segment offsets and displacements are 16-bits. This restriction limits the size of a segment that can be addressed to 64 KBytes. When the 32-bit address-size attribute is in force, segment offsets and displacements are 32-bits, allowing segments of up to 4 GBytes to be addressed.

The default operand-size attribute and/or address-size attribute can be overridden for a particular instruction by adding an operand-size and/or address-size prefix to an instruction (see "Instruction Prefixes" in Chapter 2 of the *Intel Architecture Software Developer's Manual, Volume 3*). The effect of this prefix applies only to the instruction it is attached to.

Table 3-1 shows effective operand size and address size (when executing in protected mode) depending on the settings of the B flag and the operand-size and address-size prefixes.

Table 3-1. Effective Operand- and Address-Size Attributes

D Flag in Code Segment Descriptor	0	0	0	0	1	1	1	1
Operand-Size Prefix 66H	N	N	Y	Y	N	N	Y	Y
Address-Size Prefix 67H	N	Y	N	Y	N	Y	N	Y
Effective Operand Size	16	16	32	32	32	32	16	16
Effective Address Size	16	32	16	32	32	16	32	16

NOTES:

Y Yes, this instruction prefix is present.

N No, this instruction prefix is not present.

4

Procedure Calls, Interrupts, and Exceptions

CHAPTER 4 PROCEDURE CALLS, INTERRUPTS, AND EXCEPTIONS

This chapter describes the facilities in the Intel Architecture for executing calls to procedures or subroutines. It also describes how interrupts and exceptions are handled from the perspective of an application programmer.

4.1. PROCEDURE CALL TYPES

The processor supports procedure calls in two different ways:

- CALL and RET instructions.
- ENTER and LEAVE instructions, in conjunction with the CALL and RET instructions.

Both of these procedure call mechanisms use the procedure stack, commonly referred to simply as "the stack," to save the state of the calling procedure, pass parameters to the called procedure, and store local variables for the currently executing procedure.

The processor's facilities for handling interrupts and exceptions are similar to those used by the CALL and RET instructions.

4.2. STACK

The stack (see Figure 4-1) is a contiguous array of memory locations. It is contained in a segment and identified by the segment selector in the SS register. (When using the flat memory model, the stack can be located anywhere in the linear address space for the program.) A stack can be up to 4 gigabytes long, the maximum size of a segment.

The next available memory location on the stack is called the top of stack. At any given time, the stack pointer (contained in the ESP register) gives the address (that is the offset from the base of the SS segment) of the top of the stack.

Items are placed on the stack using the PUSH instruction and removed from the stack using the POP instruction. When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack. When an item is popped off the stack, the processor reads the item from the top of stack, then increments the ESP register. In this manner, the stack grows **down** in memory (towards lesser addresses) when items are pushed on the stack and shrinks **up** (towards greater addresses) when the items are popped from the stack.

A program or operating system/executive can set up many stacks. For example, in multitasking systems, each task can be given its own stack. The number of stacks in a system is limited by the maximum number of segments and the available physical memory. When a system sets up many

stacks, only one stack—the **current stack**—is available at a time. The current stack is the one contained in the segment referenced by the SS register.

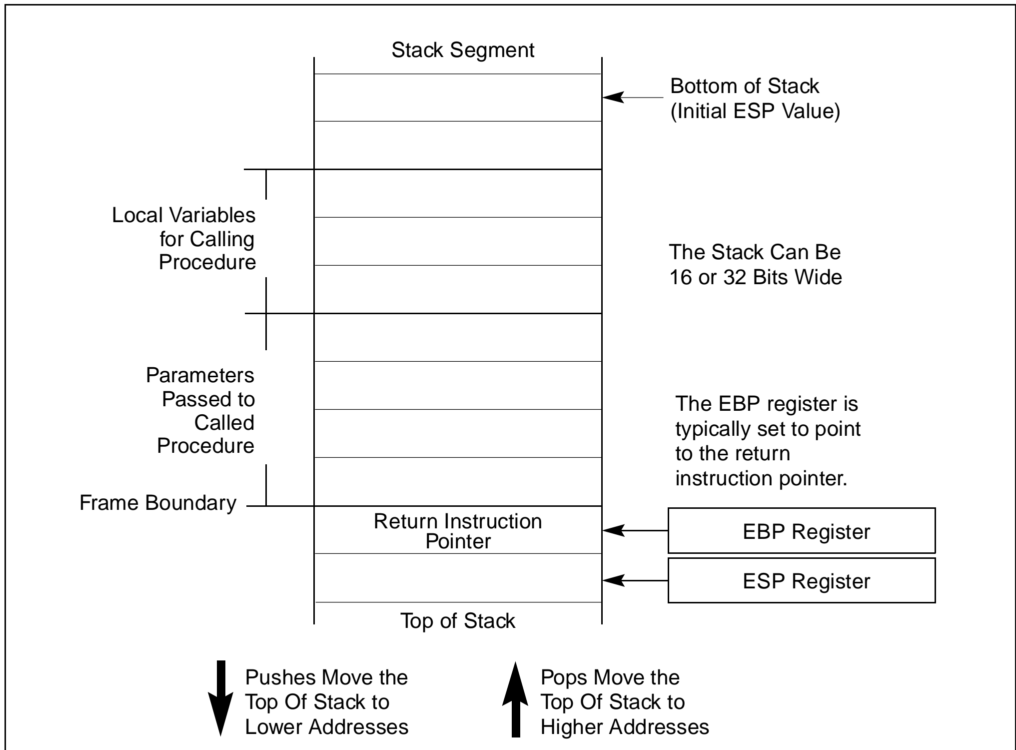


Figure 4-1. Stack Structure

The processor references the SS register automatically for all stack operations. For example, when the ESP register is used as a memory address, it automatically points to an address in the current stack. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.

4.2.1. Setting Up a Stack

To set a stack and establish it as the current stack, the program or operating system/executive must do the following:

1. Establish a stack segment.
2. Load the segment selector for the stack segment into SS register using a MOV, POP, or LSS instruction.

3. Load the stack pointer for the stack into the ESP register using a MOV, POP, or LSS instruction. (The LSS instruction can be used to load the SS and ESP registers in one operation.)

See “Segment Descriptors” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 3*, for information on how to set up a segment descriptor and segment limits for a stack segment.

4.2.2. Stack Alignment

The stack pointer for a stack segment should be aligned on 16-bit (word) or 32-bit (double-word) boundaries, depending on the width of the stack segment. The D flag in the segment descriptor for the current code segment sets the stack-segment width (see “Segment Descriptors” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 3*). The PUSH and POP instructions use the D flag to determine how much to decrement or increment the stack pointer on a push or pop operation, respectively. When the stack width is 16 bits, the stack pointer is incremented or decremented in 16-bit increments; when the width is 32 bits, the stack pointer is incremented or decremented in 32-bit increments. If a 16-bit value is pushed onto a 32-bit wide stack, the value is automatically padded with zeros out to 32 bits.

The processor does not check stack pointer alignment. It is the responsibility of the programs, tasks, and system procedures running on the processor to maintain proper alignment of stack pointers. Misaligning a stack pointer can cause serious performance degradation and in some instances program failures.

4.2.3. Address-Size Attributes for Stack Accesses

Instructions that use the stack implicitly (such as the PUSH and POP instructions) have two address-size attributes each of either 16 or 32 bits. This is because they always have the implicit address of the top of the stack, and they may also have an explicit memory address (for example, PUSH Array1[EBX]). The attribute of the explicit address is determined by the D flag of the current code segment and the presence or absence of the 67H address-size prefix, as usual.

The address-size attribute of the top of the stack determines whether SP or ESP is used for the stack access. Stack operations with an address-size attribute of 16 use the 16-bit SP stack pointer register and can use a maximum stack address of FFFFH; stack operations with a address-size attribute of 32 bits use the 32-bit ESP register and can use a maximum address of FFFFFFFFH. The default address-size attribute for data segments used as stacks is controlled by the B flag of the segment’s descriptor. When this flag is clear, the default address-size attribute is 16; when the flag is set, the address-size attribute is 32.

4.2.4. Procedure Linking Information

The processor provides two pointers for linking of procedures: the stack-frame base pointer and the return instruction pointer. When used in conjunction with a standard software procedure-call technique, these pointers permit reliable and coherent linking of procedures

4.2.4.1. STACK-FRAME BASE POINTER

The stack is typically divided into frames. Each stack frame can then contain local variables, parameters to be passed to another procedure, and procedure linking information. The stack-frame base pointer (contained in the EBP register) identifies a fixed reference point within the stack frame for the called procedure. To use the stack-frame base pointer, the called procedure typically copies the contents of the ESP register into the EBP register prior to pushing any local variables on the stack. The stack-frame base pointer then permits easy access to data structures passed on the stack, to the return instruction pointer, and to local variables added to the stack by the called procedure.

Like the ESP register, the EBP register automatically points to an address in the current stack segment (that is, the segment specified by the current contents of the SS register).

4.2.4.2. RETURN INSTRUCTION POINTER

Prior to branching to the first instruction of the called procedure, the CALL instruction pushes the address in the EIP register onto the current stack. This address is then called the return-instruction pointer and it points to the instruction where execution of the calling procedure should resume following a return from the called procedure. Upon returning from a called procedure, the RET instruction pops the return-instruction pointer from the stack back into the EIP register. Execution of the calling procedure then resumes.

The processor does not keep track of the location of the return-instruction pointer. It is thus up to the programmer to insure that stack pointer is pointing to the return-instruction pointer on the stack, prior to issuing a RET instruction. A common way to reset the stack pointer to the point to the return-instruction pointer is to move the contents of the EBP register into the ESP register. If the EBP register is loaded with the stack pointer immediately following a procedure call, it should point to the return instruction pointer on the stack.

The processor does not require that the return instruction pointer point back to the calling procedure. Prior to executing the RET instruction, the return instruction pointer can be manipulated in software to point to any address in the current code segment (near return) or another code segment (far return). Performing such an operation, however, should be undertaken very cautiously, using only well defined code entry points.

4.3. CALLING PROCEDURES USING CALL AND RET

The CALL instructions allows control transfers to procedures within the current code segment (**near call**) and in a different code segment (**far call**). Near calls usually provide access to local procedures within the currently running program or task. Far calls are usually used to access operating system procedures or procedures in a different task. See “CALL—Call Procedure” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 2*, for a detailed description of the CALL instruction.

The RET instruction also allows near and far returns to match the near and far versions of the CALL instruction. In addition, the RET instruction allows a program to increment the stack pointer on a return to release parameters from the stack. The number of bytes released from the

stack is determined by an optional argument (*n*) to the RET instruction. See “RET—Return from Procedure” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 2*, for a detailed description of the RET instruction.

4.3.1. Near CALL and RET Operation

When executing a near call, the processor does the following (see Figure 4-4):

1. Pushes the current value of the EIP register on the stack.
2. Loads the offset of the called procedure in the EIP register.
3. Begins execution of the called procedure.

When executing a near return, the processor performs these actions:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. (If the RET instruction has an optional *n* argument.) Increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
3. Resumes execution of the calling procedure.

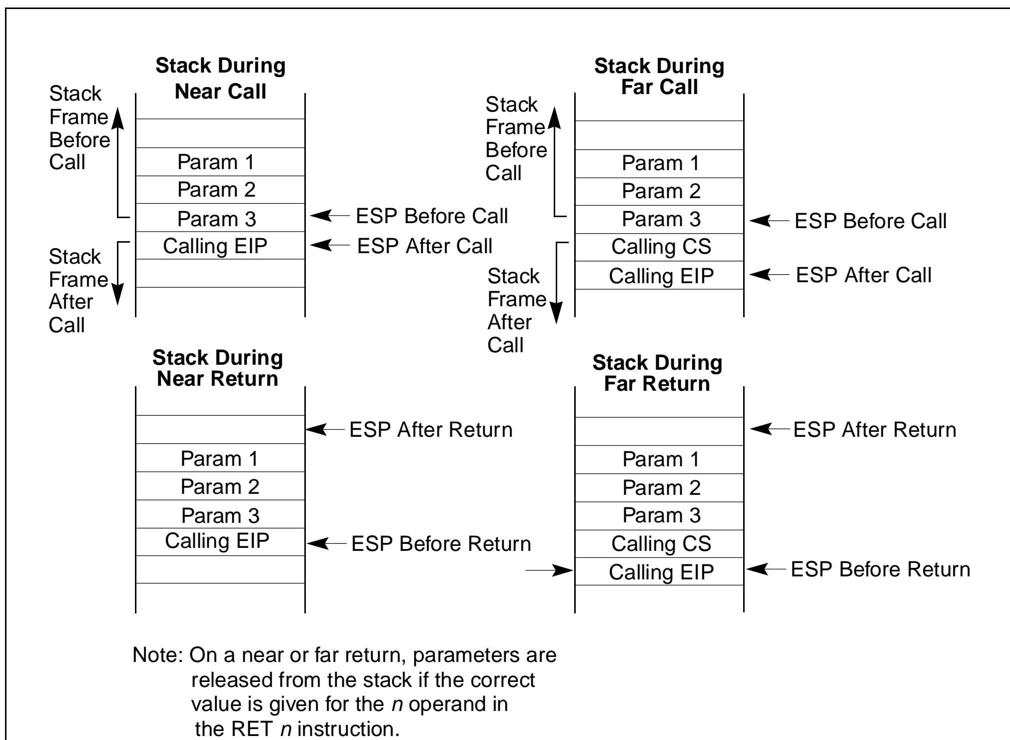


Figure 4-2. Stack on Near and Far Calls

4.3.2. Far CALL and RET Operation

When executing a far call, the processor performs these actions (see Figure 4-4):

1. Pushes current value of the CS register on the stack.
2. Pushes the current value of the EIP register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the EIP register.
5. Begins execution of the called procedure.

When executing a far return, the processor does the following:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. Pops the top-of-stack value (the segment selector for the code segment being returned to) into the CS register.
3. (If the RET instruction has an optional *n* argument.) Increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
4. Resumes execution of the calling procedure.

4.3.3. Parameter Passing

Parameters can be passed between procedures in any of three ways: through general-purpose registers, in an argument list, or on the stack.

4.3.3.1. PASSING PARAMETERS THROUGH THE GENERAL-PURPOSE REGISTERS

The processor does not save the state of the general-purpose registers on procedure calls. A calling procedure can thus pass up to six parameter to the called procedure by copying the parameters into any of these registers (except the ESP and EBP registers) prior to executing the CALL instruction. The called procedure can likewise pass parameters back to the calling procedure through general-purpose registers.

4.3.3.2. PASSING PARAMETERS ON THE STACK

To pass a large number of parameters to the called procedure, the parameters can be placed on the stack, in the stack frame for the calling procedure. Here, it is useful to use the stack-frame base pointer (in the EBP register) to make a frame boundary for easy access to the parameters.

The stack can also be used to pass parameters back from the called procedure to the calling procedure.

4.3.3.3. PASSING PARAMETERS IN AN ARGUMENT LIST

An alternate method of passing a larger number of parameters (or a data structure) to the called procedure is to place the parameters in an argument list in one of the data segments in memory. A pointer to the argument list can then be passed to the called procedure through a general-purpose register or the stack. Parameters can also be passed back to the calling procedure in this same manner.

4.3.4. Saving Procedure State Information

The processor does not save the contents of the general-purpose registers, segment registers, or the EFLAGS register on a procedure call. A calling procedure should explicitly save the values in any of the general-purpose registers that it will need when it resumes execution after a return. These values can be saved on the stack or in memory in one of the data segments.

The PUSHA and POPA instruction facilitates saving and restoring the contents of the general-purpose registers. PUSHA pushes the values in all the general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (the value prior to executing the PUSHA instruction), EBP, ESI, and EDI. The POPA instruction pops all the register values saved with a PUSHA instruction (except the ESI value) from the stack to their respective registers.

If a called procedure changes the state of any of the segment registers explicitly, it should restore them to their former value before executing a return to the calling procedure.

If a calling procedure needs to maintain the state of the EFLAGS register it can save and restore all or part of the register using the PUSHF/PUSHFD and POPF/POPFD instructions. The PUSHF instruction pushes the lower word of the EFLAGS register on the stack, while the PUSHFD instruction pushes the entire register. The POPF instruction pops a word from the stack into the lower word of the EFLAGS register, while the POPFD instruction pops a double word from the stack into the register.

4.3.5. Calls to Other Privilege Levels

The Intel Architecture’s protection mechanism recognizes four privilege levels, numbered from 0 to 3, where greater numbers mean lesser privileges. The primary reason to use these privilege levels is to improve the reliability of operating systems. For example, Figure 4-3 shows how privilege levels can be interpreted as rings of protection.

In this example, the highest privilege level 0 (at the center of the diagram) is used for segments that contain the most critical code modules in the system, usually the kernel of an operating system. The outer rings (with progressively lower privileges) are used for segments that contain code modules for less critical software.

Code modules in lower privilege segments can only access modules operating at higher privilege segments by means of a tightly controlled and protected interface called a **gate**. Attempts to access higher privilege segments without going through a protection gate and without having sufficient access rights causes a general-protection exception (#GP) to be generated.

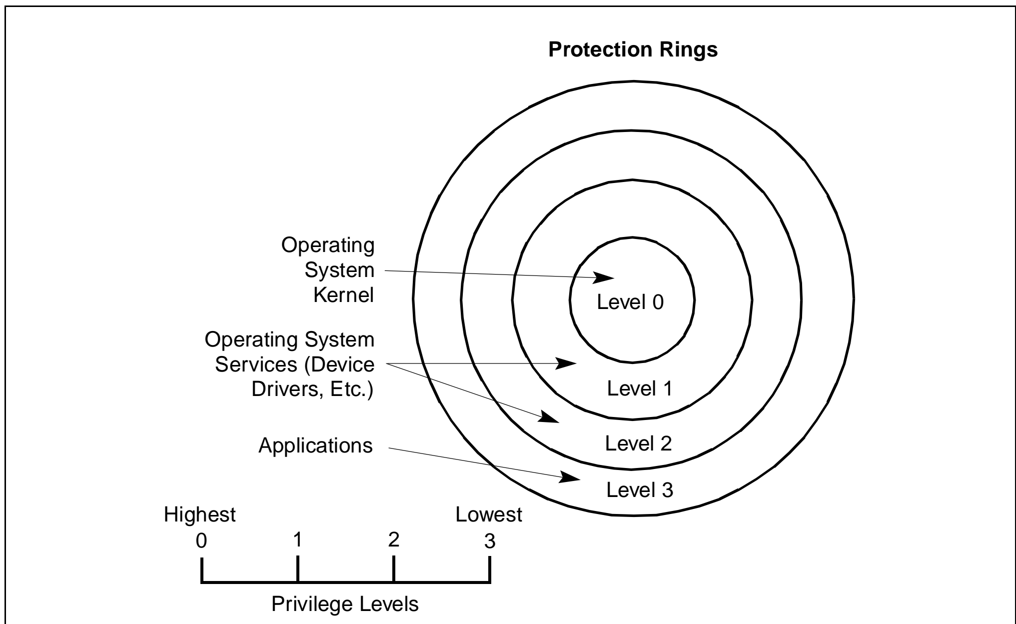


Figure 4-3. Protection Rings

If an operating system or executive uses this multilevel protection mechanism, a call to a procedure that is in a more privileged protection level than the calling procedure is handled in a similar manner as a far call (see Section 4.3.2., “Far CALL and RET Operation”). The differences are as follows:

- The segment selector provided in the CALL instruction references a special data structure called a **call gate descriptor**. Among other things, the call gate descriptor provides the following:
  - Access rights information.
  - The segment selector for the code segment of the called procedure.
  - An offset into the code segment (that is, the instruction pointer for the called procedure).
- The processor switches to a new stack to execute the called procedure. Each privilege level has its own stack. The segment selector and stack pointer for the privilege level 3 stack are stored in the SS and ESP registers, respectively, and are automatically saved when a call to a more privileged level occurs. The segment selectors and stack pointers for the privilege level 2, 1, and 0 stacks are stored in a system segment called the task state segment (TSS).

The use of a call gate and the TSS during a stack switch are transparent to the calling procedure, except when a general-protection exception is raised.

4.3.6. CALL and RET Operation Between Privilege Levels

When making a call to a more privileged protection level, the processor does the following (see Figure 4-4):

1. Performs an access rights check (privilege check).
2. Temporarily saves (internally) the current contents of the SS, ESP, CS, and EIP registers.

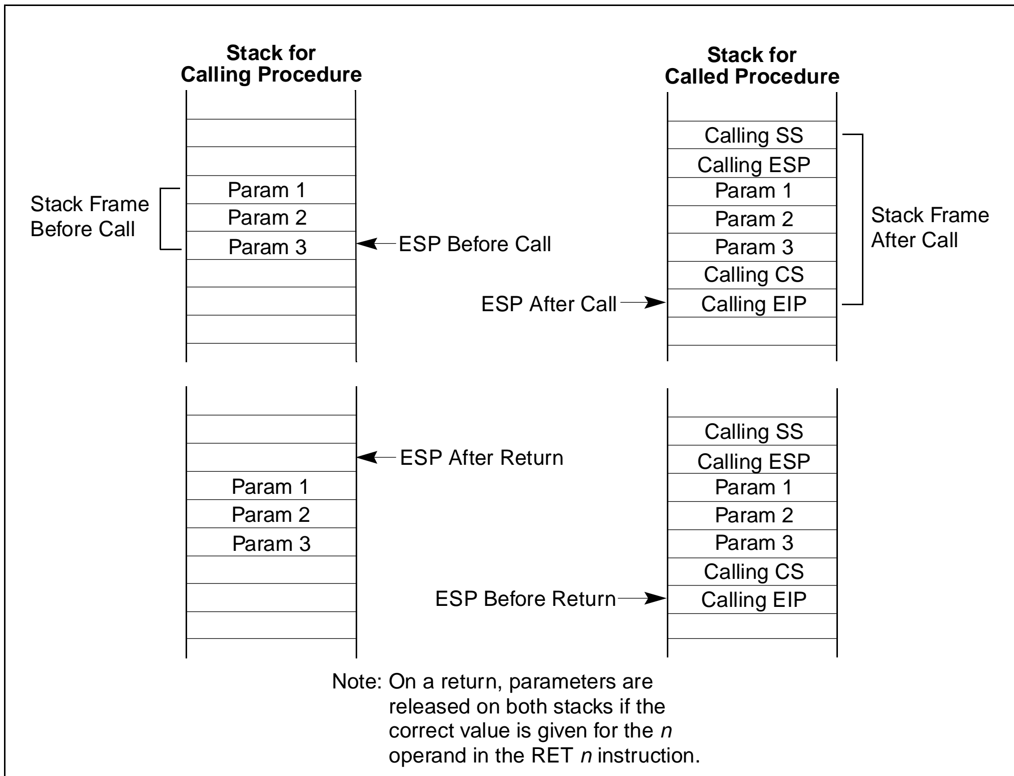


Figure 4-4. Stack Switch on a Call to a Different Privilege Level

3. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
4. Pushes the temporarily saved SS and ESP values for the calling procedure's stack onto the new stack.
5. Copies the parameters from the calling procedure's stack to the new stack. (A value in the call gate descriptor determines how many parameters to copy to the new stack.)
6. Pushes the temporarily saved CS and EIP values for the calling procedure to the new stack.

7. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively.
8. Begins execution of the called procedure at the new privilege level.

When executing a return from the privileged procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the call.
3. (If the RET instruction has an optional *n* argument.) Increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack. If the call gate descriptor specifies that one or more parameters be copied from one stack to the other, a RET *n* instruction must be used to release the parameters from both stacks. Here, the *n* operand specifies the number of bytes occupied on each stack by the parameters. On a return, the processor increments ESP by *n* for each stack to step over (effectively remove) these parameters from the stacks.
4. Restores the SS and ESP registers to their values prior to the call, which causes a switch back to the stack of the calling procedure.
5. (If the RET instruction has an optional *n* argument.) Increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack (see explanation in step 3).
6. Resumes execution of the calling procedure.

See Chapter 4, *Protection*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for detailed information on calls to privileged levels and the call gate descriptor.

4.4. INTERRUPTS AND EXCEPTIONS

The processor provides two mechanisms for interrupting program execution: interrupts and exceptions:

- An **interrupt** is an asynchronous event that is typically triggered by an I/O device.
- An **exception** is a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction. The Intel architecture specifies three classes of exceptions: faults, traps, and aborts.

The processor responds to interrupts and exceptions in essentially the same way. When an interrupt or exception is signaled, the processor halts execution of the current program or task and switches to a handler procedure that has been written specifically to handle the interrupt or exception condition. The processor accesses the handler procedure through an entry in the interrupt descriptor table (IDT). When the handler has completed handling the interrupt or exception, program control is returned to the interrupted program or task.

The operating system, executive, and/or device drivers normally handle interrupts and exceptions independently from application programs or tasks. Application programs can, however, access the interrupt and exception handlers incorporated in an operating system or executive

through assembly-language calls. The remainder of this section gives a brief overview of the processor's interrupt and exception handling mechanism. See Chapter 5, *Interrupt and Exception Handling* in the *Intel Architecture Software Developer's Manual, Volume 3*, for a detailed description of this mechanism.

The Intel Architecture defines 16 predefined interrupts and exceptions and 224 user defined interrupts, which are associated with entries in the IDT. Each interrupt and exception in the IDT is identified with a number, called a **vector**. Table 4-1 lists the interrupts and exceptions with entries in the IDT and their respective vector numbers. Vectors 0 through 8, 10 through 14, and 16 through 18 are the predefined interrupts and exceptions, and vectors 32 through 255 are the user-defined interrupts, called **maskable interrupts**.

Note that the processor defines several additional interrupts that do not point to entries in the IDT; the most notable of these interrupts is the SMI interrupt. See "Exception and Interrupt Vectors" in Chapter 5 of the *Intel Architecture Software Developer's Manual, Volume 3*, for more information about the interrupts and exceptions that the Intel Architecture supports.

When the processor detects an interrupt or exception, it does one of the following things:

- Executes an implicit call to a handler procedure.
- Executes an implicit call to a handler task.

4.4.1. Call and Return Operation for Interrupt or Exception Handling Procedures

A call to an interrupt or exception handler procedure is similar to a procedure call to another protection level (see Section 4.3.6., "CALL and RET Operation Between Privilege Levels"). Here, the interrupt vector references one of two kinds of gates: an **interrupt gate** or a **trap gate**. Interrupt and trap gates are similar to call gates in that they provide the following information:

- Access rights information.
- The segment selector for the code segment that contains the handler procedure.
- An offset into the code segment to the first instruction of the handler procedure.

The difference between an interrupt gate and a trap gate is as follows. If an interrupt or exception handler is called through an interrupt gate, the processor clears the interrupt enable (IF) flag in the EFLAGS register to prevent subsequent interrupts from interfering with the execution of the handler. When a handler is called through a trap gate, the state of the IF flag is not changed.

If the code segment for the handler procedure has the same privilege level as the currently executing program or task, the handler procedure uses the current stack; if the handler executes at a more privileged level, the processor switches to the stack for the handler's privilege level.

Table 4-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9		CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		(Intel reserved. Do not use.)	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19-31		(Intel reserved. Do not use.)	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

1. The UD2 instruction was introduced in the Pentium® Pro processor.
2. Intel Architecture processors after the Intel386™ processor do not generate this exception.
3. This exception was introduced in the Intel486™ processor.
4. This exception was introduced in the Pentium processor and enhanced in the Pentium Pro processor.

If no stack switch occurs, the processor does the following when calling an interrupt or exception handler (see Figure 4-5):

1. Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
2. Pushes an error code (if appropriate) on the stack.
3. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
4. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
5. Begins execution of the handler procedure at the new privilege level.

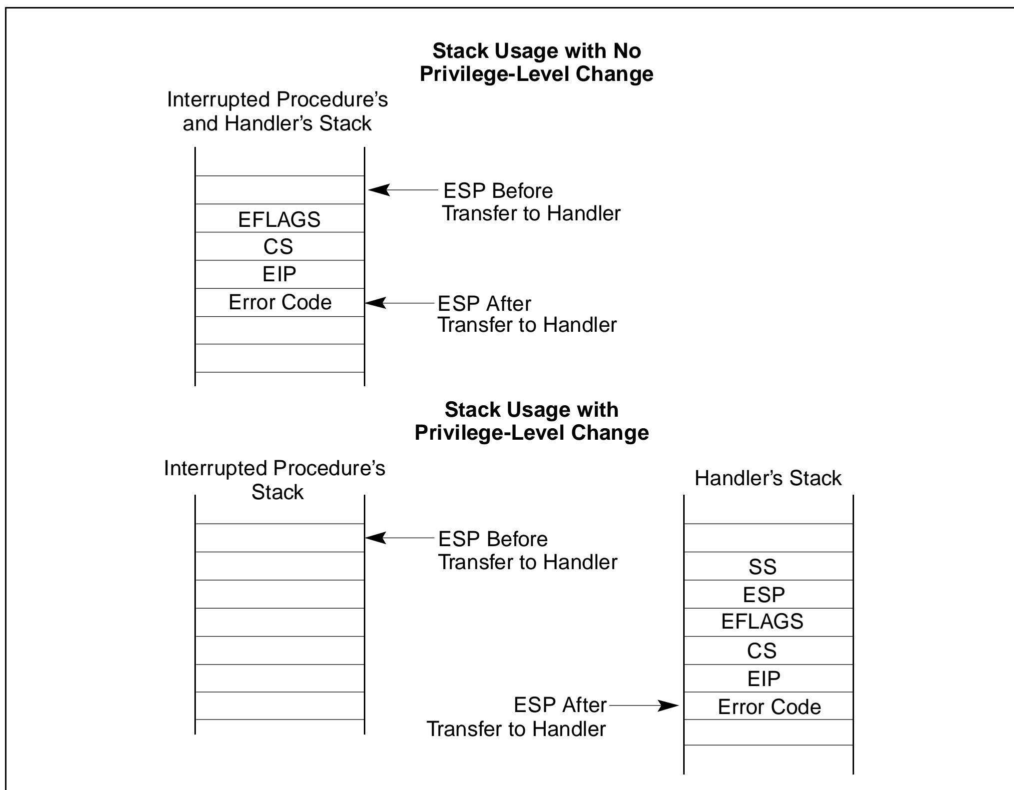


Figure 4-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines

If a stack switch does occur, the processor does the following:

1. Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS, and EIP registers.

2. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
3. Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack onto the new stack.
4. Pushes an error code on the new stack (if appropriate).
5. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
6. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
7. Begins execution of the handler procedure at the new privilege level.

A return from an interrupt or exception handler is initiated with the IRET instruction. The IRET instruction is similar to the far RET instruction, except that it also restores the contents of the EFLAGS register for the interrupted procedure:

When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions:

1. Restores the CS and EIP registers to their values prior to the interrupt or exception.
2. Restores the EFLAGS register.
3. Increments the stack pointer appropriately
4. Resumes execution of the interrupted procedure.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the interrupt or exception.
3. Restores the EFLAGS register.
4. Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.
5. Resumes execution of the interrupted procedure.

4.4.2. Calls to Interrupt or Exception Handler Tasks

Interrupt and exception handler routines can also be executed in a separate task. Here, an interrupt or exception causes a task switch to a handler task. The handler task is given its own address space and (optionally) can execute at a higher protection level than application programs or tasks.

The switch to the handler task is accomplished with an implicit task call that references a **task gate descriptor**. The task gate provides access to the address space for the handler task. As part

of the task switch, the processor saves complete state information for the interrupted program or task. Upon returning from the handler task, the state of the interrupted program or task is restored and execution continues. See Chapter 5, *Interrupt and Exception Handling*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for a detailed description of the processor's mechanism for handling interrupts and exceptions through handler tasks.

4.4.3. Interrupt and Exception Handling in Real-Address Mode

When operating in real-address mode, the processor responds to an interrupt or exception with an implicit far call to an interrupt or exception handler. The processor uses the interrupt or exception vector number as an index into an interrupt table. The interrupt table contains instruction pointers to the interrupt and exception handler procedures.

The processor saves the state of the EFLAGS register, the EIP register, the CS register, and an optional error code on the stack before switching to the handler procedure.

A return from the interrupt or exception handler is carried out with the IRET instruction.

See Chapter 15, *8086 Emulation*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on handling interrupts and exceptions in real-address mode.

4.4.4. INT *n*, INTO, INT 3, and BOUND Instructions

The INT *n*, INTO, INT 3, and BOUND instructions allow a program or task to explicitly call an interrupt or exception handler. The INT *n* instruction uses an interrupt vector as an argument, which allows a program to call any interrupt handler.

The INTO instruction explicitly calls the overflow exception (#OF) handler if the overflow flag (OF) in the EFLAGS register is set. The OF flag indicates overflow on arithmetic instructions, but it does not automatically raise an overflow exception. An overflow exception can only be raised explicitly in either of the following ways:

- Execute the INTO instruction.
- Test the OF flag and execute the INT *n* instruction with an argument of 4 (the vector number of the overflow exception) if the flag is set.

Both the methods of dealing with overflow conditions allow a program to test for overflow at specific places in the instruction stream.

The INT 3 instruction explicitly calls the breakpoint exception (#BP) handler.

The BOUND instruction explicitly calls the BOUND-range exceeded exception (#BR) handler if an operand is found to be not within predefined boundaries in memory. This instruction is provided for checking references to arrays and other data structures. Like the overflow exception, the BOUND-range exceeded exception can only be raised explicitly with the BOUND instruction or the INT *n* instruction with an argument of 5 (the vector number of the bounds-check exception). The processor does not implicitly perform bounds checks and raise the BOUND-range exceeded exception.

4.5. PROCEDURE CALLS FOR BLOCK-STRUCTURED LANGUAGES

The Intel Architecture supports an alternate method of performing procedure calls with the ENTER (enter procedure) and LEAVE (leave procedure) instructions. These instructions automatically create and release, respectively, stack frames for called procedures. The stack frames have predefined spaces for local variables and the necessary pointers to allow coherent returns from called procedures. They also allow scope rules to be implemented so that procedures can access their own local variables and some number of other variables located in other stack frames.

The ENTER and LEAVE instructions offer two benefits:

- They provide machine-language support for implementing block-structured languages, such as C and Pascal.
- They simplify procedure entry and exit in compiler-generated code.

4.5.1. ENTER Instruction

The ENTER instruction creates a stack frame compatible with the scope rules typically used in block-structured languages. In block-structured languages, the scope of a procedure is the set of variables to which it has access. The rules for scope vary among languages. They may be based on the nesting of procedures, the division of the program into separately compiled files, or some other modularization scheme.

The ENTER instruction has two operands. The first specifies the number of bytes to be reserved on the stack for dynamic storage for the procedure being called. Dynamic storage is the memory allocated for variables created when the procedure is called, also known as automatic variables. The second parameter is the lexical nesting level (from 0 to 31) of the procedure. The nesting level is the depth of a procedure in a hierarchy of procedure calls. The lexical level is unrelated to either the protection privilege level or to the I/O privilege level of the currently running program or task.

The ENTER instruction in the following example, allocates 2K bytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame for this procedure.

```
ENTER 2048,3
```

The lexical nesting level determines the number of stack frame pointers to copy into the new stack frame from the preceding frame. A stack frame pointer is a doubleword used to access the variables of a procedure. The set of stack frame pointers used by a procedure to access the variables of other procedures is called the display. The first doubleword in the display is a pointer to the previous stack frame. This pointer is used by a LEAVE instruction to undo the effect of an ENTER instruction by discarding the current stack frame.

After the ENTER instruction creates the display for a procedure, it allocates the dynamic local variables for the procedure by decrementing the contents of the ESP register by the number of bytes specified in the first parameter. This new value in the ESP register serves as the initial top-of-stack for all PUSH and POP operations within the procedure.

To allow a procedure to address its display, the ENTER instruction leaves the EBP register pointing to the first doubleword in the display. Because stacks grow down, this is actually the doubleword with the highest address in the display. Data manipulation instructions that specify the EBP register as a base register automatically address locations within the stack segment instead of the data segment.

The ENTER instruction can be used in two ways: nested and non-nested. If the lexical level is 0, the non-nested form is used. The non-nested form pushes the contents of the EBP register on the stack, copies the contents of the ESP register into the EBP register, and subtracts the first operand from the contents of the ESP register to allocate dynamic storage. The non-nested form differs from the nested form in that no stack frame pointers are copied. The nested form of the ENTER instruction occurs when the second parameter (lexical level) is not zero.

The following pseudo code shows the formal definition of the ENTER instruction. STORAGE is the number of bytes of dynamic storage to allocate for local variables, and LEVEL is the lexical nesting level.

```
PUSH EBP;
FRAME_PTR ← ESP;
IF LEVEL > 0
   THEN
        DO (LEVEL – 1) times
             EBP ← EBP – 4;
             PUSH Pointer(EBP); (* doubleword pointed to by EBP *)
        OD;
   PUSH FRAME_PTR;
FI;
EBP ← FRAME_PTR;
ESP ← ESP – STORAGE;
```

The main procedure (in which all other procedures are nested) operates at the highest lexical level, level 1. The first procedure it calls operates at the next deeper lexical level, level 2. A level 2 procedure can access the variables of the main program, which are at fixed locations specified by the compiler. In the case of level 1, the ENTER instruction allocates only the requested dynamic storage on the stack because there is no previous display to copy.

A procedure which calls another procedure at a lower lexical level gives the called procedure access to the variables of the caller. The ENTER instruction provides this access by placing a pointer to the calling procedure's stack frame in the display.

A procedure which calls another procedure at the same lexical level should not give access to its variables. In this case, the ENTER instruction copies only that part of the display from the calling procedure which refers to previously nested procedures operating at higher lexical levels. The new stack frame does not include the pointer for addressing the calling procedure’s stack frame.

The ENTER instruction treats a re-entrant procedure as a call to a procedure at the same lexical level. In this case, each succeeding iteration of the re-entrant procedure can address only its own variables and the variables of the procedures within which it is nested. A re-entrant procedure always can address its own variables; it does not require pointers to the stack frames of previous iterations.

By copying only the stack frame pointers of procedures at higher lexical levels, the ENTER instruction makes certain that procedures access only those variables of higher lexical levels, not those at parallel lexical levels (see Figure 4-6).

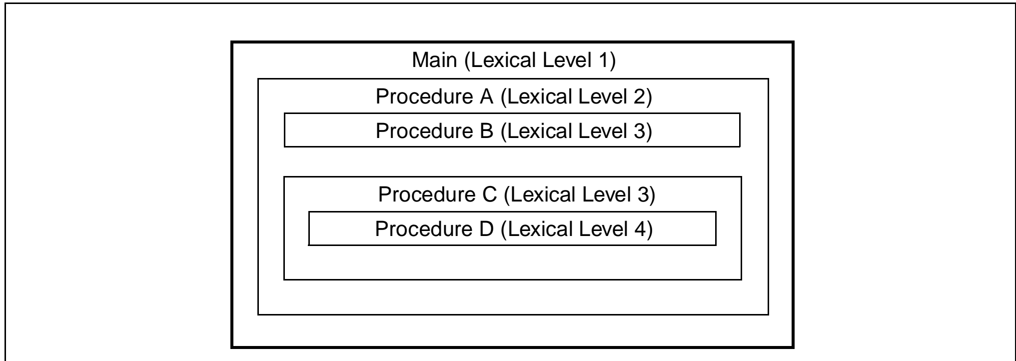


Figure 4-6. Nested Procedures

Block-structured languages can use the lexical levels defined by ENTER to control access to the variables of nested procedures. In Figure 4-6, for example, if procedure A calls procedure B which, in turn, calls procedure C, then procedure C will have access to the variables of the MAIN procedure and procedure A, but not those of procedure B because they are at the same lexical level. The following definition describes the access to variables for the nested procedures in Figure 4-6.

1. MAIN has variables at fixed locations.
2. Procedure A can access only the variables of MAIN.
3. Procedure B can access only the variables of procedure A and MAIN. Procedure B cannot access the variables of procedure C or procedure D.
4. Procedure C can access only the variables of procedure A and MAIN. procedure C cannot access the variables of procedure B or procedure D.
5. Procedure D can access the variables of procedure C, procedure A, and MAIN. Procedure D cannot access the variables of procedure B.

In Figure 4-7, an ENTER instruction at the beginning of the MAIN procedure creates three doublewords of dynamic storage for MAIN, but copies no pointers from other stack frames. The first doubleword in the display holds a copy of the last value in the EBP register before the ENTER instruction was executed. The second doubleword holds a copy of the contents of the EBP register following the ENTER instruction. After the instruction is executed, the EBP register points to the first doubleword pushed on the stack, and the ESP register points to the last doubleword in the stack frame.

When MAIN calls procedure A, the ENTER instruction creates a new display (see Figure 4-8). The first doubleword is the last value held in MAIN's EBP register. The second doubleword is a pointer to MAIN's stack frame which is copied from the second doubleword in MAIN's display.

This happens to be another copy of the last value held in MAIN's EBP register. Procedure A can access variables in MAIN because MAIN is at level 1. Therefore the base address for the dynamic storage used in MAIN is the current address in the EBP register, plus four bytes to account for the saved contents of MAIN's EBP register. All dynamic variables for MAIN are at fixed, positive offsets from this value.

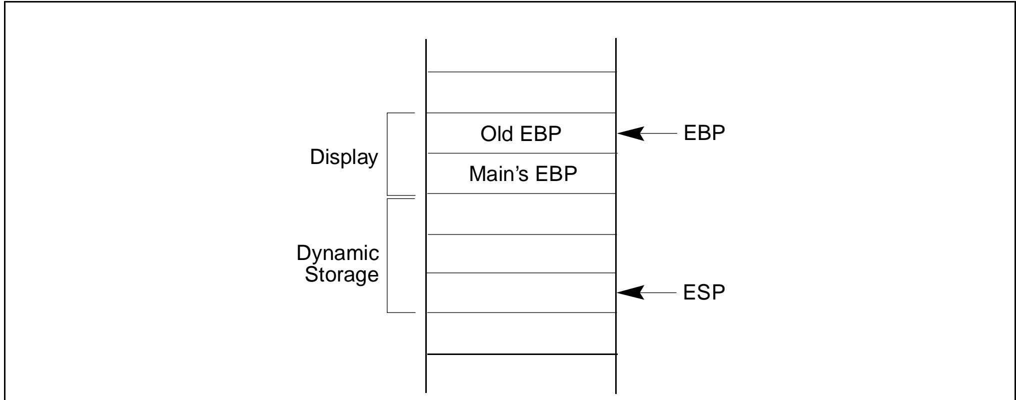


Figure 4-7. Stack Frame after Entering the MAIN Procedure

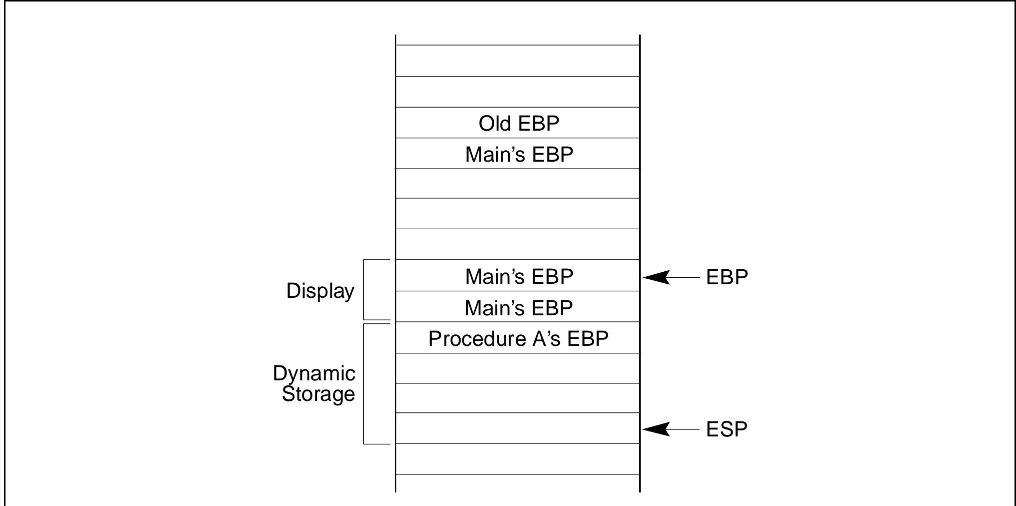


Figure 4-8. Stack Frame after Entering Procedure A

When procedure A calls procedure B, the ENTER instruction creates a new display (see Figure 4-9). The first doubleword holds a copy of the last value in procedure A's EBP register. The second and third doublewords are copies of the two stack frame pointers in procedure A's display. Procedure B can access variables in procedure A and MAIN by using the stack frame pointers in its display.

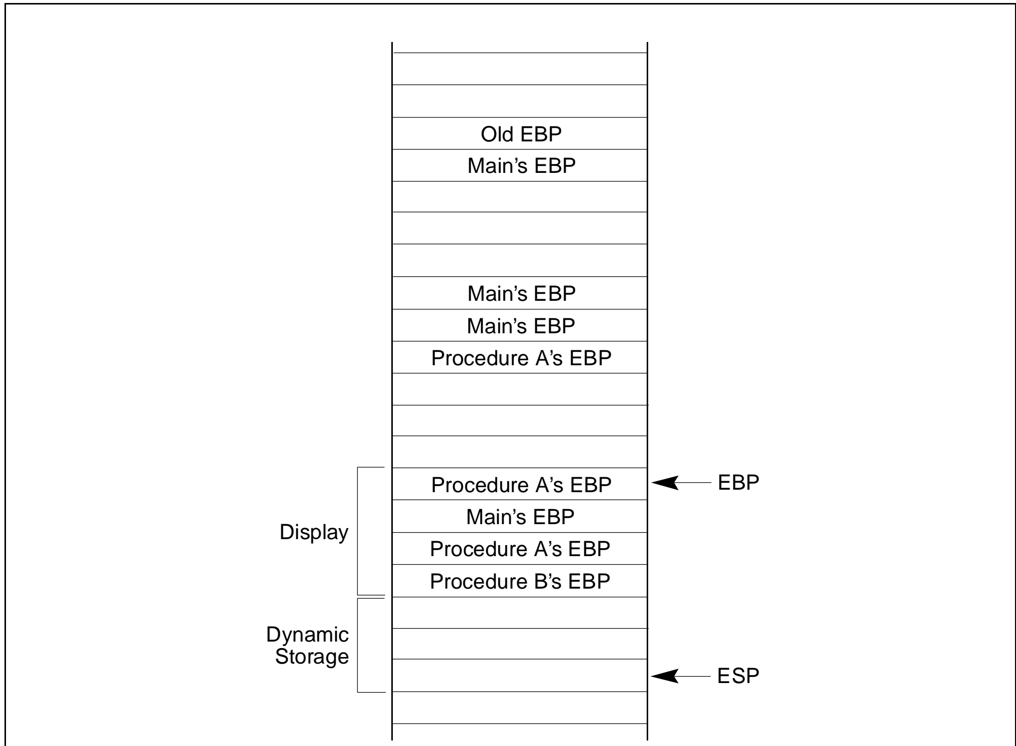


Figure 4-9. Stack Frame after Entering Procedure B

When procedure B calls procedure C, the ENTER instruction creates a new display for procedure C (see Figure 4-10). The first doubleword holds a copy of the last value in procedure B's EBP register. This is used by the LEAVE instruction to restore procedure B's stack frame. The second and third doublewords are copies of the two stack frame pointers in procedure A's display. If procedure C were at the next deeper lexical level from procedure B, a fourth doubleword would be copied, which would be the stack frame pointer to procedure B's local variables.

Note that procedure B and procedure C are at the same level, so procedure C is not intended to access procedure B's variables. This does not mean that procedure C is completely isolated from procedure B; procedure C is called by procedure B, so the pointer to the returning stack frame is a pointer to procedure B's stack frame. In addition, procedure B can pass parameters to procedure C either on the stack or through variables global to both procedures (that is, variables in the scope of both procedures).

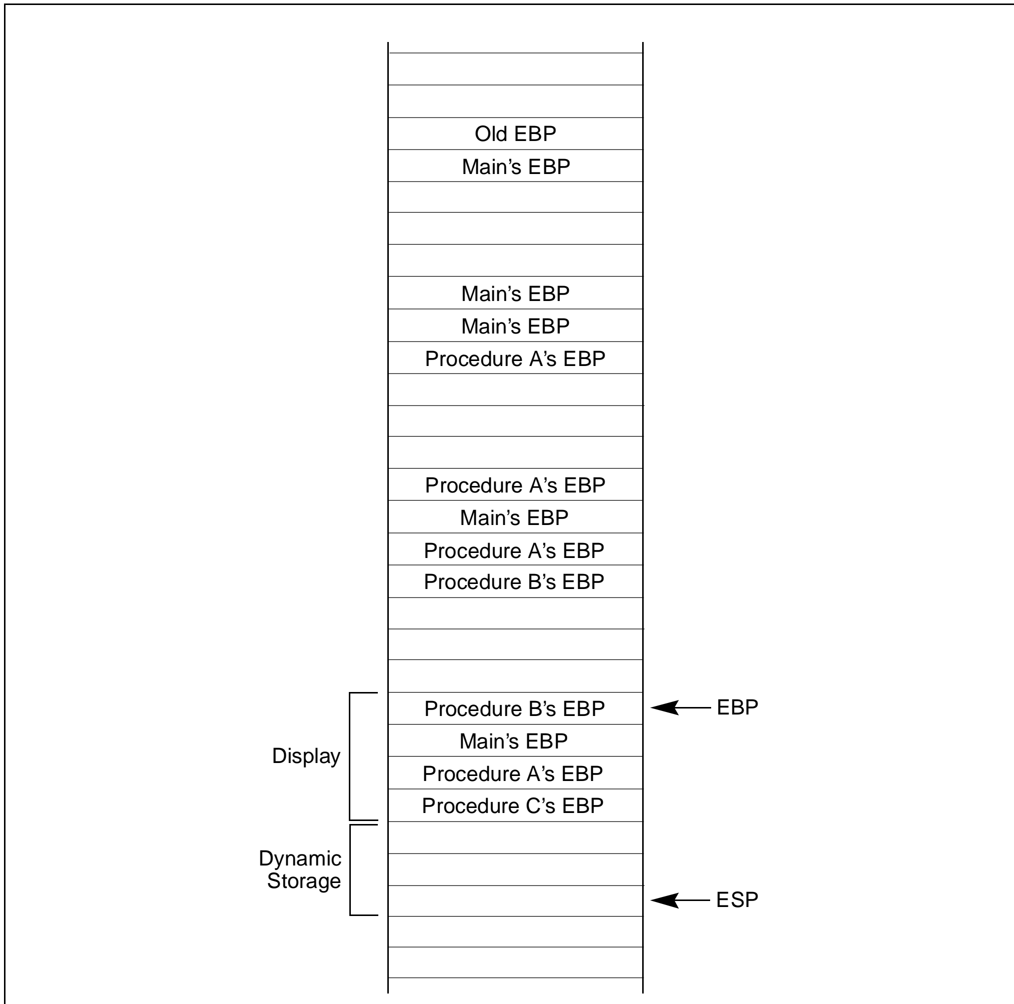


Figure 4-10. Stack Frame after Entering Procedure C

4.5.2. LEAVE Instruction

The LEAVE instruction, which does not have any operands, reverses the action of the previous ENTER instruction. The LEAVE instruction copies the contents of the EBP register into the ESP register to release all stack space allocated to the procedure. Then it restores the old value of the EBP register from the stack. This simultaneously restores the ESP register to its original value. A subsequent RET instruction then can remove any arguments and the return address pushed on the stack by the calling program for use by the procedure.

5

Data Types and Addressing Modes

CHAPTER 5
DATA TYPES AND ADDRESSING MODES

This chapter describes data types and addressing modes available to programmers of the Intel Architecture processors.

5.1. FUNDAMENTAL DATA TYPES

The fundamental data types of the Intel Architecture are bytes, words, doublewords, and quadwords (see Figure 5-1). A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), and a quadword is 8 bytes (64 bits).

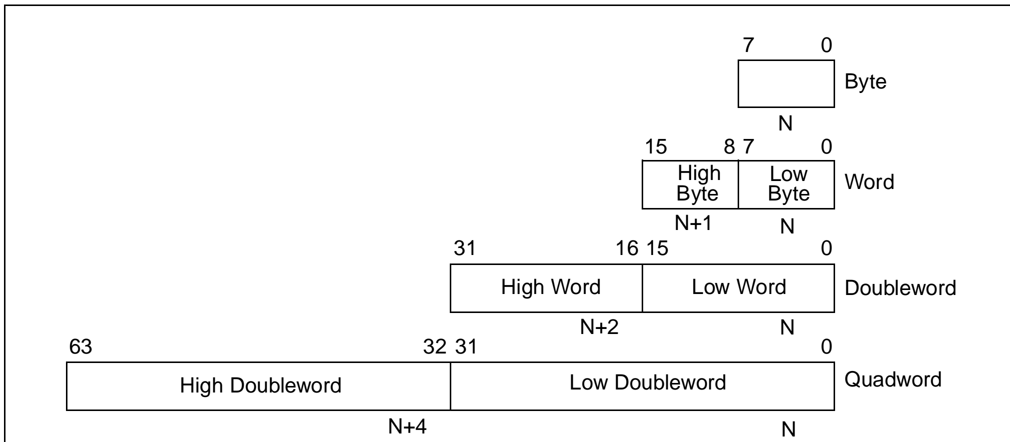


Figure 5-1. Fundamental Data Types

Figure 5-2 shows the byte order of each of the fundamental data types when referenced as operands in memory. The low byte (bits 0 through 7) of each data type occupies the lowest address in memory and that address is also the address of the operand.

5.1.1. Alignment of Words, Doublewords, and Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. (The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively.) However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; whereas,

aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles to access it; a word that starts on an odd address but does not cross a word boundary is considered aligned and can still be accessed in one bus cycle.

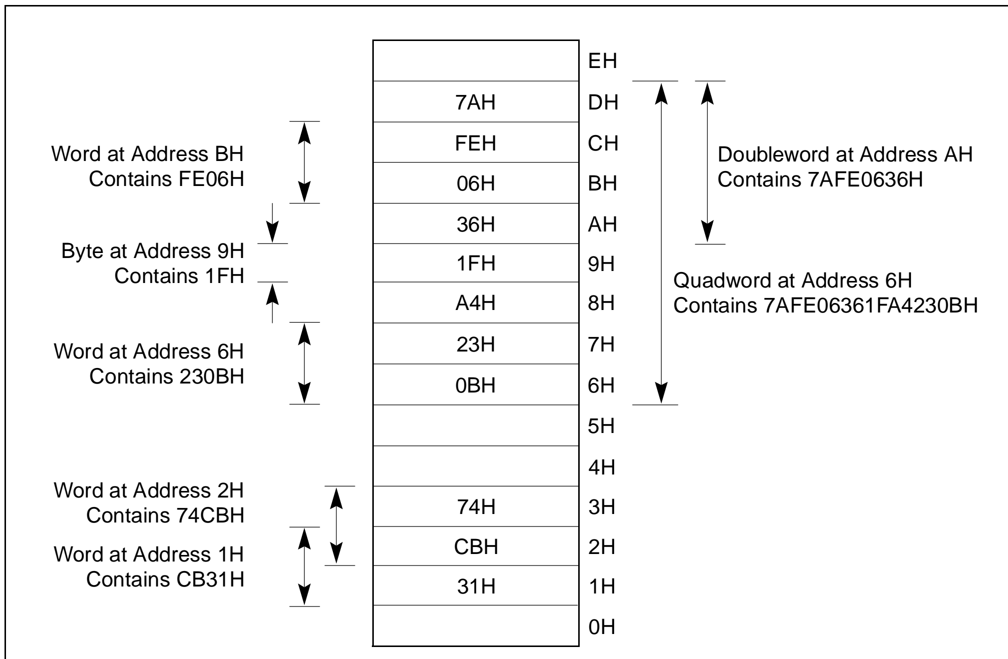


Figure 5-2. Bytes, Words, Doublewords and Quadwords in Memory

5.2. NUMERIC, POINTER, BIT FIELD, AND STRING DATA TYPES

Although bytes, words, and doublewords are the fundamental data types of the Intel Architecture, some instructions support additional interpretations of these data types to allow operations to be performed on numeric data types (signed and unsigned integers and BCD integers). See Figure 5-3. Also, some instructions recognize and operate on additional pointer, bit field, and string data types. The following sections describe these additional data types.

5.2.1. Integers

Integers are signed binary numbers held in a byte, word, or doubleword. All operations assume a two's complement representation. The sign bit is located in bit 7 in a byte integer, bit 15 in a word integer, and bit 31 in a doubleword integer. The sign bit is set for negative integers and cleared for positive integers and zero. Integer values range from –128 to +127 for a byte integer, from –32,768 to +32,767 for a word integer, and from -2^{31} to $+2^{31} - 1$ for a doubleword integer.

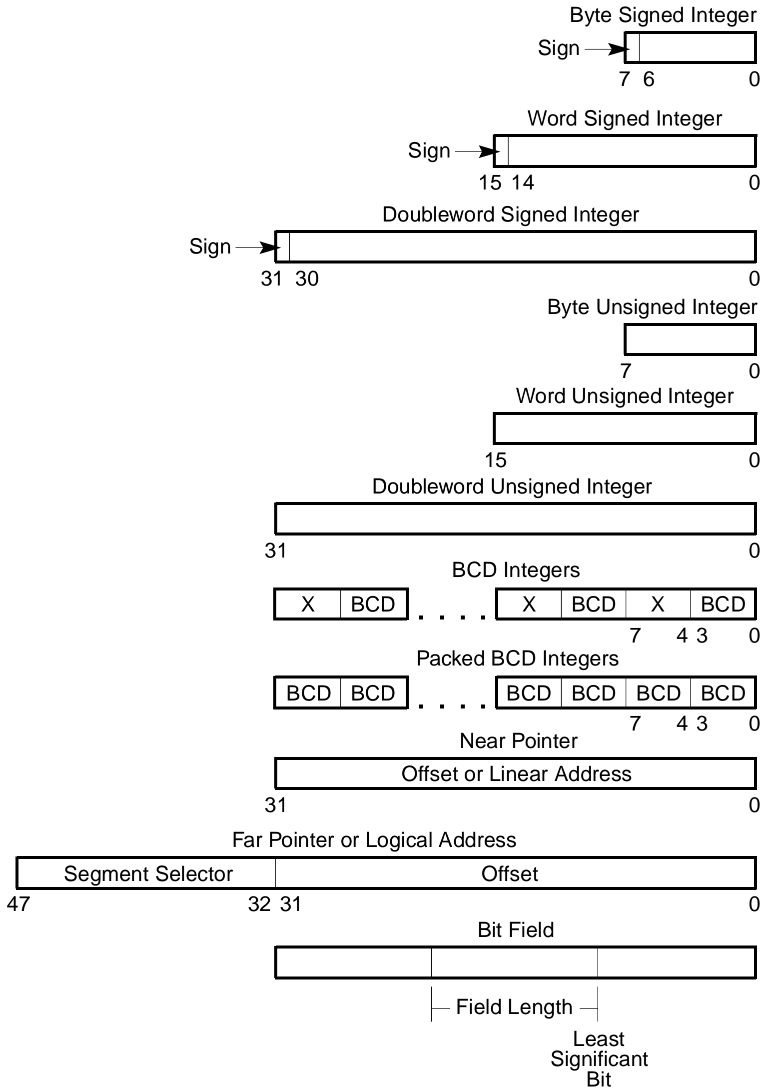


Figure 5-3. Numeric, Pointer, and Bit Field Data Types

5.2.2. Unsigned Integers

Unsigned integers are unsigned binary numbers contained in a byte, word, or doubleword. Unsigned integer values range from 0 to 255 for an unsigned byte integer, from 0 to 65,535 for an unsigned word integer, and from 0 to $2^{32} - 1$ for an unsigned doubleword integer. Unsigned integers are sometimes referred to as **ordinals**.

5.2.3. BCD Integers

Binary-coded decimal integers (BCD integers) are unsigned 4-bit integers with valid values ranging from 0 to 9. BCD integers can be unpacked (one BCD digit per byte) or packed (two BCD digits per byte). The value of an unpacked BCD integer is the binary value of the low half-byte (bits 0 through 3). The high half-byte (bits 4 through 7) can be any value during addition and subtraction, but must be zero during multiplication and division.

Packed BCD integers allow two BCD digits to be contained in one byte. Here, the digit in the high half-byte is more significant than the digit in the low half-byte.

5.2.4. Pointers

Pointers are addresses of locations in memory. The Pentium Pro processor recognizes two types of pointers: a **near pointer** (32 bits) and a **far pointer** (48 bits). A near pointer is a 32-bit offset (also called an **effective address**) within a segment. Near pointers are used for all memory references in a flat memory model or for references in a segmented model where the identity of the segment being accessed is implied. A far pointer is a 48-bit logical address, consisting of a 16-bit segment selector and a 32-bit offset. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.

5.2.5. Bit Fields

A **bit field** is a contiguous sequence of bits. It can begin at any bit position of any byte in memory and can contain up to 32 bits.

5.2.6. Strings

Strings are continuous sequences of bits, bytes, words, or doublewords. A **bit string** can begin at any bit position of any byte and can contain up to $2^{32} - 1$ bits. A **byte string** can contain bytes, words, or doublewords and can range from zero to $2^{32} - 1$ bytes (4 gigabytes).

5.2.7. Floating-Point Data Types

The processor's floating-point instructions recognize a set of real, integer, and BCD integer data types. See Section 7.4., "Floating-Point Data Types and Formats", for a description of FPU data types.

5.2.8. MMX™ Technology Data Types

Intel Architecture processors that implement the Intel MMX technology recognize a set of packed 64-bit data types. See Section 8.1.2., "MMX™ Data Types", for a description of the MMX data types.

5.3. OPERAND ADDRESSING

An Intel Architecture machine-instruction acts on zero or more operands. Some operands are specified explicitly in an instruction and others are implicit to an instruction. An operand can be located in any of the following places:

- The instruction itself (an immediate operand).
- A register.
- A memory location.
- An I/O port.

5.3.1. Immediate Operands

Some instructions use data encoded in the instruction itself as a source operand. These operands are called **immediate** operands (or simply immediates). For example, the following ADD instruction adds an immediate value of 14 to the contents of the EAX register:

```
ADD EAX, 14
```

All the arithmetic instructions (except the DIV and IDIV instructions) allow the source operand to be an immediate value. The maximum value allowed for an immediate operand varies among instructions, but can never be greater than the maximum value of an unsigned doubleword integer (2^{32}).

5.3.2. Register Operands

Source and destination operands can be located in any of the following registers, depending on the instruction being executed:

- The 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP).
- The 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP).

- The 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL).
- The segment registers (CS, DS, SS, ES, FS, and GS).
- The EFLAGS register.
- System registers, such as the global descriptor table (GDTR) or the interrupt descriptor table register (IDTR).

Some instructions (such as the DIV and MUL instructions) use quadword operands contained in a pair of 32-bit registers. Register pairs are represented with a colon separating them. For example, in the register pair EDX:EAX, EDX contains the high order bits and EAX contains the low order bits of a quadword operand.

Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the J*cc* instructions) use the state of the status flags in the EFLAGS register as condition codes for branching or other decision making operations.

The processor contains a selection of system registers that are used to control memory management, interrupt and exception handling, task management, processor management, and debugging activities. Some of these system registers are accessible by an application program, the operating system, or the executive through a set of system instructions. When accessing a system register with a system instruction, the register is generally an implied operand of the instruction.

5.3.3. Memory Operands

Source and destination operands in memory are referenced by means of a segment selector and an offset (see Figure 5-4). The segment selector specifies the segment containing the operand and the offset (the number of bytes from the beginning of the segment to the first byte of the operand) specifies the linear or effective address of the operand.

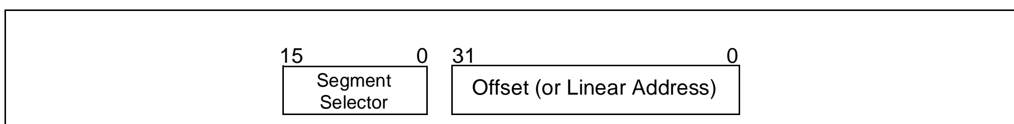


Figure 5-4. Memory Operand Address

5.3.3.1. SPECIFYING A SEGMENT SELECTOR

The segment selector can be specified either implicitly or explicitly. The most common method of specifying a segment selector is to load it in a segment register and then allow the processor to select the register implicitly, depending on the type of operation being performed. The processor automatically chooses a segment according to the rules given in Table 5-1.

Table 5-1. Default Segment Selection Rules

Type of Reference	Register Used	Segment Used	Default Selection Rule
Instructions	CS	Code Segment	All instruction fetches.
Stack	SS	Stack Segment	All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.
Local Data	DS	Data Segment	All data references, except when relative to stack or string destination.
Destination Strings	ES	Data Segment pointed to with the ES register	Destination of string instructions.

When storing data in or loading data from memory, the DS segment default can be overridden to allow other segments to be accessed. Within an assembler, the segment override is generally handled with a colon ":" operator. For example, the following MOV instruction moves a value from register EAX into the segment pointed to by the ES register. The offset into the segment is contained in the EBX register:

```
MOV ES:[EBX], EAX;
```

(At the machine level, a segment override is specified with a segment-override prefix, which is a byte placed at the beginning of an instruction.) The following default segment selections cannot be overridden:

- Instruction fetches must be made from the code segment.
- Destination strings in string instructions must be stored in the data segment pointed to by the ES register.
- Push and pop operations must always reference the SS segment.

Some instructions require a segment selector to be specified explicitly. In these cases, the 16-bit segment selector can be located in a memory location or in a 16-bit register. For example, the following MOV instruction moves a segment selector located in register BX into segment register DS:

```
MOV DS, BX
```

Segment selectors can also be specified explicitly as part of a 48-bit far pointer in memory. Here, the first doubleword in memory contains the offset and the next word contains the segment selector.

5.3.3.2. SPECIFYING AN OFFSET

The offset part of a memory address can be specified either directly as an static value (called a **displacement**) or through an address computation made up of one or more of the following components:

- Displacement—An 8-, 16-, or 32-bit value.
- Base—The value in a general-purpose register.
- Index—The value in a general-purpose register.
- Scale factor—A value of 2, 4, or 8 that is multiplied by the index value.

The offset which results from adding these components is called an **effective address**. Each of these components can have either a positive or negative (2s complement) value, with the exception of the scaling factor. Figure 5-5 shows all the possible ways that these components can be combined to create an effective address in the selected segment.

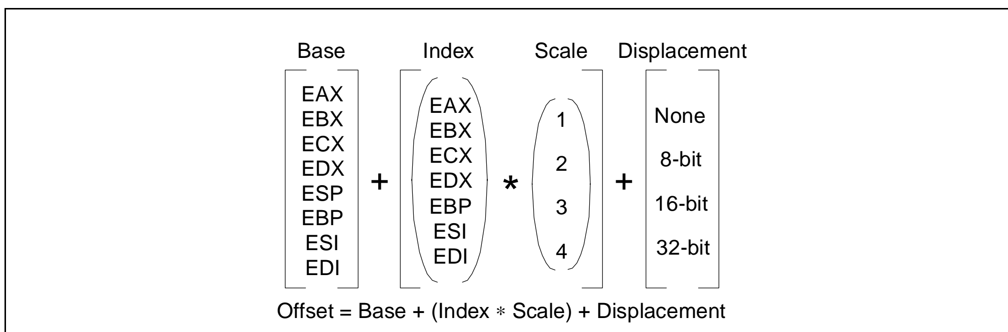


Figure 5-5. Offset (or Effective Address) Computation

The uses of general-purpose registers as base or index components are restricted in the following manner:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

The base, index, and displacement components can be used in any combination, and any of these components can be null. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language. The following addressing modes suggest uses for common combinations of address components.

Displacement

A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

Base

A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.

Base + Displacement

A base register and a displacement can be used together for two distinct purposes:

- As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
- To access a field of a record—The base register holds the address of the beginning of the record, while the displacement is an static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

(Index * Scale) + Displacement

This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

Base + Index + Displacement

Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).

Base + (Index * Scale) + Displacement

Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

5.3.3.3. ASSEMBLER AND COMPILER ADDRESSING MODES

At the machine-code level, the selected combination of displacement, base register, index register, and scale factor is encoded in an instruction. All assemblers permit a programmer to use any of the allowable combinations of these addressing components to address operands. High-level language (HLL) compilers will select an appropriate combination of these components based on the HHL construct a programmer defines.

5.3.4. I/O Port Addressing

The processor supports an I/O address space that contains up to 65,536 8-bit I/O ports. Ports that are 16-bit and 32-bit may also be defined in the I/O address space. An I/O port can be addressed with either an immediate operand or a value in the DX register. See Chapter 9, *Input/Output*, for more information about I/O port addressing.

6

Instruction Set Summary

CHAPTER 6 INSTRUCTION SET SUMMARY

This chapter lists all the instructions in the Intel Architecture instruction set, divided into three functional groups: integer, floating-point, and system. It also briefly describes each of the integer instructions.

Brief descriptions of the floating-point instructions are given in Chapter 7, *Floating-Point Unit*; brief descriptions of the system instructions are given in the *Intel Architecture Software Developer's Manual, Volume 3*.

Detailed descriptions of all the Intel Architecture instructions are given in *Intel Architecture Software Developer's Manual, Volume 2*. Included in this volume are a description of each instruction's encoding and operation, the effect of an instruction on the EFLAGS flags, and the exceptions an instruction may generate.

6.1. NEW INTEL ARCHITECTURE INSTRUCTIONS

The following sections give the Intel Architecture instructions that were new in the MMX Technology and in the Pentium Pro, Pentium, and Intel486 processors.

6.1.1. New Instructions Introduced with the MMX™ Technology

The Intel MMX technology introduced a new set of instructions to the Intel Architecture, designed to enhance the performance of multimedia applications. These instructions are recognized by all Intel Architecture processors that implement the MMX technology. The MMX instructions are listed in Section 6.2.2., "MMX™ Technology Instructions".

6.1.2. New Instructions in the Pentium® Pro Processor

The following instructions are new in the Pentium Pro processor:

- CMOV*cc*—Conditional move (see Section 6.3.1.2., "Conditional Move Instructions").
- FCMOV*cc*—Floating-point conditional move on condition-code flags in EFLAGS register (see Section 7.5.3., "Data Transfer Instructions").
- FCOMI/FCOMIP/FUCOMI/FUCOMIP—Floating-point compare and set condition-code flags in EFLAGS register (see Section 7.5.6., "Comparison and Classification Instructions").
- RDPMC—Read performance monitoring counters (see "RDPMC—Read Performance-Monitoring Counters" in Chapter 3 of the *Intel Architecture Software Developer's Manual,*

Volume 2). (This instruction is also available in all Pentium® processors that implement the MMX™ technology.)

- UD2—Undefined instruction (see Section 6.15.4., "No-Operation and Undefined Instructions").

6.1.3. New Instructions in the Pentium® Processor

The following instructions are new in the Pentium processor:

- CMPXCHG8B (compare and exchange 8 bytes) instruction.
- CPUID (CPU identification) instruction. (This instruction was introduced in the Pentium® processor and added to later versions of the Intel486™ processor.)
- RDTSC (read time-stamp counter) instruction.
- RDMSR (read model-specific register) instruction.
- WRMSR (write model-specific register) instruction.
- RSM (resume from SMM) instruction.

The form of the MOV instruction used to access the test registers has been removed on the Pentium and future Intel Architecture processors.

6.1.4. New Instructions in the Intel486™ Processor

The following instructions are new in the Intel486 processor:

- BSWAP (byte swap) instruction.
- XADD (exchange and add) instruction.
- CMPXCHG (compare and exchange) instruction.
- INVD (invalidate cache) instruction.
- WBINVD (write-back and invalidate cache) instruction.
- INVLPG (invalidate TLB entry) instruction.

6.2. INSTRUCTION SET LIST

This section lists all the Intel Architecture instructions divided into three major groups: integer, MMX technology, floating-point, and system instructions. For each instruction, the mnemonic and descriptive names are given. When two or more mnemonics are given (for example, CMOVA/CMOVNBE), they represent different mnemonics for the same instruction opcode. Assemblers support redundant mnemonics for some instructions to make it easier to read code listings. For instance, CMOVA (Conditional move if above) and CMOVNBE (Conditional move is not below or equal) represent the same condition.

6.2.1. Integer Instructions

Integer instructions perform the integer arithmetic, logic, and program flow control operations that programmers commonly use to write application and system software to run on an Intel Architecture processor. In the following sections, the integer instructions are divided into several instruction subgroups.

6.2.1.1. DATA TRANSFER INSTRUCTIONS

MOV	Move
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater
CMOVC	Conditional move if carry
CMOVNC	Conditional move if not carry
CMOVO	Conditional move if overflow
CMOVNO	Conditional move if not overflow
CMOVS	Conditional move if sign (negative)
CMOVNS	Conditional move if not sign (non-negative)
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd
XCHG	Exchange

BSWAP	Byte swap
XADD	Exchange and add
CMPXCHG	Compare and exchange
CMPXCHG8B	Compare and exchange 8 bytes
PUSH	Push onto stack
POP	Pop off of stack
PUSHA/PUSHAD	Push general-purpose registers onto stack
POPA/POPAD	Pop general-purpose registers from stack
IN	Read from a port
OUT	Write to a port
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register
MOVSX	Move and sign extend
MOVZX	Move and zero extend

6.2.1.2. BINARY ARITHMETIC INSTRUCTIONS

ADD	Integer add
ADC	Add with carry
SUB	Subtract
SBB	Subtract with borrow
IMUL	Signed multiply
MUL	Unsigned multiply
IDIV	Signed divide
DIV	Unsigned divide
INC	Increment
DEC	Decrement
NEG	Negate
CMP	Compare

6.2.1.3. DECIMAL ARITHMETIC

DAA	Decimal adjust after addition
DAS	Decimal adjust after subtraction

AAA	ASCII adjust after addition
AAS	ASCII adjust after subtraction
AAM	ASCII adjust after multiplication
AAD	ASCII adjust before division

6.2.1.4. LOGIC INSTRUCTIONS

AND	And
OR	Or
XOR	Exclusive or
NOT	Not

6.2.1.5. SHIFT AND ROTATE INSTRUCTIONS

SAR	Shift arithmetic right
SHR	Shift logical right
SAL/SHL	Shift arithmetic left/Shift logical left
SHRD	Shift right double
SHLD	Shift left double
ROR	Rotate right
ROL	Rotate left
RCR	Rotate through carry right
RCL	Rotate through carry left

6.2.1.6. BIT AND BYTE INSTRUCTIONS

BT	Bit test
BTS	Bit test and set
BTR	Bit test and reset
BTC	Bit test and complement
BSF	Bit scan forward
BSR	Bit scan reverse
SETE/SETZ	Set byte if equal/Set byte if zero
SETNE/SETNZ	Set byte if not equal/Set byte if not zero

SETA/SETNBE	Set byte if above/Set byte if not below or equal
SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry
SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry
SETBE/SETNA	Set byte if below or equal/Set byte if not above
SETG/SETNLE	Set byte if greater/Set byte if not less or equal
SETGE/SETNL	Set byte if greater or equal/Set byte if not less
SETL/SETNGE	Set byte if less/Set byte if not greater or equal
SETLE/SETNG	Set byte if less or equal/Set byte if not greater
SETS	Set byte if sign (negative)
SETNS	Set byte if not sign (non-negative)
SETO	Set byte if overflow
SETNO	Set byte if not overflow
SETPE/SETP	Set byte if parity even/Set byte if parity
SETPO/SETNP	Set byte if parity odd/Set byte if not parity
TEST	Logical compare

6.2.1.7. CONTROL TRANSFER INSTRUCTIONS

JMP	Jump
JE/JZ	Jump if equal/Jump if zero
JNE/JNZ	Jump if not equal/Jump if not zero
JA/JNBE	Jump if above/Jump if not below or equal
JAE/JNB	Jump if above or equal/Jump if not below
JB/JNAE	Jump if below/Jump if not above or equal
JBE/JNA	Jump if below or equal/Jump if not above
JG/JNLE	Jump if greater/Jump if not less or equal
JGE/JNL	Jump if greater or equal/Jump if not less
JL/JNGE	Jump if less/Jump if not greater or equal
JLE/JNG	Jump if less or equal/Jump if not greater
JC	Jump if carry
JNC	Jump if not carry

JO	Jump if overflow
JNO	Jump if not overflow
JS	Jump if sign (negative)
JNS	Jump if not sign (non-negative)
JPO/JNP	Jump if parity odd/Jump if not parity
JPE/JP	Jump if parity even/Jump if parity
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero
LOOP	Loop with ECX counter
LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal
CALL	Call procedure
RET	Return
IRET	Return from interrupt
INT	Software interrupt
INTO	Interrupt on overflow
BOUND	Detect value out of range
ENTER	High-level procedure entry
LEAVE	High-level procedure exit

6.2.1.8. STRING INSTRUCTIONS

MOVS/MOVSB	Move string/Move byte string
MOVS/MOVSW	Move string/Move word string
MOVS/MOVSD	Move string/Move doubleword string
CMPS/CMPSB	Compare string/Compare byte string
CMPS/CMPSW	Compare string/Compare word string
CMPS/CMPSD	Compare string/Compare doubleword string
SCAS/SCASB	Scan string/Scan byte string
SCAS/SCASW	Scan string/Scan word string
SCAS/SCASD	Scan string/Scan doubleword string
LODS/LODSB	Load string/Load byte string
LODS/LODSW	Load string/Load word string

LODS/LODSD	Load string/Load doubleword string
STOS/STOSB	Store string/Store byte string
STOS/STOSW	Store string/Store word string
STOS/STOSD	Store string/Store doubleword string
REP	Repeat while ECX not zero
REPE/REPZ	Repeat while equal/Repeat while zero
REPNE/REPNZ	Repeat while not equal/Repeat while not zero
INS/INSB	Input string from port/Input byte string from port
INS/INSW	Input string from port/Input word string from port
INS/INSD	Input string from port/Input doubleword string from port
OUTS/OUTSB	Output string to port/Output byte string to port
OUTS/OUTSW	Output string to port/Output word string to port
OUTS/OUTSD	Output string to port/Output doubleword string to port

6.2.1.9. FLAG CONTROL INSTRUCTIONS

STC	Set carry flag
CLC	Clear the carry flag
CMC	Complement the carry flag
CLD	Clear the direction flag
STD	Set direction flag
LAHF	Load flags into AH register
SAHF	Store AH register into flags
PUSHF/PUSHFD	Push EFLAGS onto stack
POPF/POPFD	Pop EFLAGS from stack
STI	Set interrupt flag
CLI	Clear the interrupt flag

6.2.1.10. SEGMENT REGISTER INSTRUCTIONS

LDS	Load far pointer using DS
LES	Load far pointer using ES
LFS	Load far pointer using FS

LGS	Load far pointer using GS
LSS	Load far pointer using SS

6.2.1.11. MISCELLANEOUS INSTRUCTIONS

LEA	Load effective address
NOP	No operation
UB2	Undefined instruction
XLAT/XLATB	Table lookup translation
CPUID	Processor Identification

6.2.2. MMX™ Technology Instructions

The MMX instructions execute on those Intel Architecture processors that implement the Intel MMX technology. These instructions operate on packed-byte, packed-word, packed-double-word, and quadword operands. As with the integer instructions, the following list of MMX instructions is divided into subgroups.

6.2.2.1. MMX™ DATA TRANSFER INSTRUCTIONS

MOVD	Move doubleword
MOVQ	Move quadword

6.2.2.2. MMX™ CONVERSION INSTRUCTIONS

PACKSSWB	Pack words into bytes with signed saturation
PACKSSDW	Pack doublewords into words with signed saturation
PACKUSWB	Pack words into bytes with unsigned saturation
PUNPCKHBW	Unpack high-order bytes from words
PUNPCKHWD	Unpack high-order words from doublewords
PUNPCKHDQ	Unpack high-order doublewords from quadword
PUNPCKLBW	Unpack low-order bytes from words
PUNPCKLWD	Unpack low-order words from doublewords
PUNPCKLDQ	Unpack low-order doublewords from quadword

6.2.2.3. MMX™ PACKED ARITHMETIC INSTRUCTIONS

PADDB	Add packed bytes
PADDW	Add packed words
PADDD	Add packed doublewords
PADDSB	Add packed bytes with saturation
PADDSW	Add packed words with saturation
PADDUSB	Add packed unsigned bytes with saturation
PADDUSW	Add packed unsigned words with saturation
PSUBB	Subtract packed bytes
PSUBW	Subtract packed words
PSUBD	Subtract packed doublewords
PSUBSB	Subtract packed bytes with saturation
PSUBSW	Subtract packed words with saturation
PSUBUSB	Subtract packed unsigned bytes with saturation
PSUBUSW	Subtract packed unsigned words with saturation
PMULHW	Multiply packed words and store high result
PMULLW	Multiply packed words and store low result
PMADDWD	Multiply and add packed words

6.2.2.4. MMX™ COMPARISON INSTRUCTIONS

PCMPEQB	Compare packed bytes for equal
PCMPEQW	Compare packed words for equal
PCMPEQD	Compare packed doublewords for equal
PCMPGTB	Compare packed bytes for greater than
PCMPGTW	Compare packed words for greater than
PCMPGTD	Compare packed doublewords for greater than

6.2.2.5. MMX™ LOGIC INSTRUCTIONS

PAND	Bitwise logical and
PANDN	Bitwise logical and not
POR	Bitwise logical or
PXOR	Bitwise logical exclusive or

6.2.2.6. MMX™ SHIFT AND ROTATE INSTRUCTIONS

PSLLW	Shift packed words left logical
PSLLD	Shift packed doublewords left logical
PSLLQ	Shift packed quadword left logical
PSRLW	Shift packed words right logical
PSRLD	Shift packed doublewords right logical
PSRLQ	Shift packed quadword right logical
PSRAW	Shift packed words right arithmetic
PSRAD	Shift packed doublewords right arithmetic

6.2.2.7. MMX™ STATE MANAGEMENT

EMMS	Empty MMX state

6.2.3. Floating-Point Instructions

The floating-point instructions are those that are executed by the processor's floating-point unit (FPU). These instructions operate on floating-point (real), extended integer, and binary-coded decimal (BCD) operands. As with the integer instructions, the following list of floating-point instructions is divided into subgroups.

6.2.3.1. DATA TRANSFER

FLD	Load real
FST	Store real
FSTP	Store real and pop
FILD	Load integer
FIST	Store integer
FISTP	Store integer and pop
FBLD	Load BCD
FBSTP	Store BCD and pop
FXCH	Exchange registers
FCMOVE	Floating-point conditional move if equal
FCMOVNE	Floating-point conditional move if not equal
FCMOVB	Floating-point conditional move if below

FCMOVBE	Floating-point conditional move if below or equal
FCMOVNB	Floating-point conditional move if not below
FCMOVNBE	Floating-point conditional move if not below or equal
FCMOVU	Floating-point conditional move if unordered
FCMOVNU	Floating-point conditional move if not unordered

6.2.3.2. BASIC ARITHMETIC

FADD	Add real
FADDP	Add real and pop
FIADD	Add integer
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Subtract integer
FSUBR	Subtract real reverse
FSUBRP	Subtract real reverse and pop
FISUBR	Subtract integer reverse
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Multiply integer
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Divide integer
FDIVR	Divide real reverse
FDIVRP	Divide real reverse and pop
FIDIVR	Divide integer reverse
FPREM	Partial remainder
FPREMI	IEEE Partial remainder
FABS	Absolute value
FCHS	Change sign
FRNDINT	Round to integer
FSCALE	Scale by power of two

FSQRT	Square root
FXTRACT	Extract exponent and significand

6.2.3.3. COMPARISON

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FUCOM	Unordered compare real
FUCOMP	Unordered compare real and pop
FUCOMPP	Unordered compare real and pop twice
FICOM	Compare integer
FICOMP	Compare integer and pop
FCOMI	Compare real and set EFLAGS
FUCOMI	Unordered compare real and set EFLAGS
FCOMIP	Compare real, set EFLAGS, and pop
FUCOMIP	Unordered compare real, set EFLAGS, and pop
FTST	Test real
FXAM	Examine real

6.2.3.4. TRANSCENDENTAL

FSIN	Sine
FCOS	Cosine
FSINCOS	Sine and cosine
FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$y*\log_2 x$
FYL2XP1	$y*\log_2(x+1)$

6.2.3.5. LOAD CONSTANTS

FLD1	Load +1.0
FLDZ	Load +0.0

FLDPI	Load π
FLDL2E	Load $\log_2 e$
FLDLN2	Load $\log_e 2$
FLDL2T	Load $\log_2 10$
FLDLG2	Load $\log_{10} 2$

6.2.3.6. FPU CONTROL

FINCSTP	Increment FPU register stack pointer
FDECSTP	Decrement FPU register stack pointer
FFREE	Free floating-point register
FINIT	Initialize FPU after checking error conditions
FNINIT	Initialize FPU without checking error conditions
FCLEX	Clear floating-point exception flags after checking for error conditions
FNCLEX	Clear floating-point exception flags without checking for error conditions
FSTCW	Store FPU control word after checking error conditions
FNSTCW	Store FPU control word without checking error conditions
FLDCW	Load FPU control word
FSTENV	Store FPU environment after checking error conditions
FNSTENV	Store FPU environment without checking error conditions
FLDENV	Load FPU environment
FSAVE	Save FPU state after checking error conditions
FNSAVE	Save FPU state without checking error conditions
FRSTOR	Restore FPU state
FSTSW	Store FPU status word after checking error conditions
FNSTSW	Store FPU status word without checking error conditions
WAIT/FWAIT	Wait for FPU
FNOP	FPU no operation

6.2.4. System Instructions

The following system instructions are used to control those functions of the processor that are provided to support for operating systems and executives.

LGDT	Load global descriptor table (GDT) register
SGDT	Store global descriptor table (GDT) register
LLDT	Load local descriptor table (LDT) register
SLDT	Store local descriptor table (LDT) register
LTR	Load task register
STR	Store task register
LIDT	Load interrupt descriptor table (IDT) register
SIDT	Store interrupt descriptor table (IDT) register
MOV	Load and store control registers
LMSW	Load machine status word
SMSW	Store machine status word
CLTS	Clear the task-switched flag
ARPL	Adjust requested privilege level
LAR	Load access rights
LSL	Load segment limit
VERR	Verify segment for reading
VERW	Verify segment for writing
MOV	Load and store debug registers
INVD	Invalidate cache, no writeback
WBINVD	Invalidate cache, with writeback
INVLPG	Invalidate TLB Entry
LOCK (prefix)	Lock Bus
HLT	Halt processor
RSM	Return from system management mode (SSM)
RDMSR	Read model-specific register
WRMSR	Write model-specific register
RDPMC	Read performance monitoring counters
RDTSC	Read time stamp counter

6.3. DATA MOVEMENT INSTRUCTIONS

The data movement instructions move bytes, words, doublewords, or quadwords both between memory and the processor's registers and between registers. These instructions are divided into four groups:

- General-purpose data movement.
- Exchange.
- Stack manipulation.
- Type-conversion.

6.3.1. General-Purpose Data Movement Instructions

The MOV (move) and CMOVcc (conditional move) instructions transfer data between memory and registers or between registers.

6.3.1.1. MOVE INSTRUCTION

The MOV instruction performs basic load data and store data operations between memory and the processor's registers and data movement operations between registers. It handles data transfers along the paths listed in Table 6-1. (See "MOV—Move to/from Control Registers" and "MOV—Move to/from Debug Registers" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*, for information on moving data to and from the control and debug registers.)

The MOV instruction cannot move data from one memory location to another or from one segment register to another segment register. Memory-to-memory moves can be performed with the MOVS (string move) instruction (see Section 6.10., "String Operations").

6.3.1.2. CONDITIONAL MOVE INSTRUCTIONS

The CMOVcc instructions are a group of instructions that check the state of the status flags in the EFLAGS register and perform a move operation if the flags are in a specified state (or condition). These instructions can be used to move a 16- or 32-bit value from memory to a general-purpose register or from one general-purpose register to another. The flag state being tested for each instruction is specified with a condition code (cc) that is associated with the instruction. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction.

Table 6-1. Move Instruction Operations

Type of Data Movement	Source → Destination
From memory to a register	Memory location → General-purpose register Memory location → Segment register
From a register to memory	General-purpose register → Memory location Segment register → Memory location
Between registers	General-purpose register → General-purpose register General-purpose register → Segment register Segment register → General-purpose register General-purpose register → Control register Control register → General-purpose register General-purpose register → Debug register Debug register → General-purpose register
Immediate data to a register	Immediate → General-purpose register
Immediate data to memory	Immediate → Memory location

Table 6-4 shows the mnemonics for the CMOVcc instructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letters "CMOV" to form the mnemonics for the CMOVcc instructions. The instructions listed in Table 6-4 as pairs (for example, CMOVA/CMOVNBE) are alternate names for the same instruction. The assembler provides these alternate names to make it easier to read program listings.

The CMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF statements and the possibility of branch mispredictions by the processor.

These instructions may not be supported on some processors in the Pentium Pro processor family. Software can check if the CMOVcc instructions are supported by checking the processor's feature information with the CPUID instruction (see "CPUID—CPU Identification" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*).

6.3.1.3. EXCHANGE INSTRUCTIONS

The exchange instructions swap the contents of one or more operands and, in some cases, performs additional operations such as asserting the LOCK signal or modifying flags in the EFLAGS register.

The XCHG (exchange) instruction swaps the contents of two operands. This instruction takes the place of three MOV instructions and does not require a temporary location to save the contents of one operand location while the other is being loaded. When a memory operand is used with the XCHG instruction, the processor's LOCK signal is automatically asserted. This instruction is thus useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 7 of the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on bus locking.)

The BSWAP (byte swap) instruction reverses the byte order in a 32-bit register operand. Bit positions 0 through 7 are exchanged with 24 through 31, and bit positions 8 through 15 are exchanged with 16 through 23. Executing this instruction twice in a row leaves the register with the same

value as before. The BSWAP instruction is useful for converting between "big-endian" and "little-endian" data formats. This instruction also speeds execution of decimal arithmetic. (The XCHG instruction can be used two swap the bytes in a word.)

Table 6-2. Conditional Move Instructions

Instruction Mnemonic	Status Flag States	Condition Description
Unsigned Conditional Moves		
CMOVA/CMOVNBE	(CF or ZF)=0	Above/not below or equal
CMOVAE/CMOVNB	CF=0	Above or equal/not below
CMOVNC	CF=0	Not carry
CMOVB/CMOVNAE	CF=1	Below/not above or equal
CMOVC	CF=1	Carry
CMOVBE/CMOVNA	(CF or ZF)=1	Below or equal/not above
CMOVE/CMOVZ	ZF=1	Equal/zero
CMOVNE/CMOVNZ	ZF=0	Not equal/not zero
CMOVP/CMOVPE	PF=1	Parity/parity even
CMOVNP/CMOVPO	PF=0	Not parity/parity odd
Signed Conditional Moves		
CMOVGE/CMOVNL	(SF xor OF)=0	Greater or equal/not less
CMOVL/CMOVNGE	(SF xor OF)=1	Less/not greater or equal
CMOVLE/CMOVNG	((SF xor OF) or ZF)=1	Less or equal/not greater
CMOVO	OF=1	Overflow
CMOVNO	OF=0	Not overflow
CMOVS	SF=1	Sign (negative)
CMOVNS	SF=0	Not sign (non-negative)

The XADD (exchange and add) instruction swaps two operands and then stores the sum of the two operands in the destination operand. The status flags in the EFLAGS register indicate the result of the addition. This instruction can be combined with the LOCK prefix (see "LOCK—Assert LOCK# Signal Prefix" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*) in a multiprocessing system to allow multiple processors to execute one DO loop.

The CMPXCHG (compare and exchange) and CMPXCHG8B (compare and exchange 8 bytes) instructions are used to synchronize operations in systems that use multiple processors. The CMPXCHG instruction requires three operands: a source operand in a register, another source operand in the EAX register, and a destination operand. If the values contained in the destination operand and the EAX register are equal, the destination operand is replaced with the value of the other source operand (the value not in the EAX register). Otherwise, the original value of the destination operand is loaded in the EAX register. The status flags in the EFLAGS register

reflect the result that would have been obtained by subtracting the destination operand from the value in the EAX register.

The CMPXCHG instruction is commonly used for testing and modifying semaphores. It checks to see if a semaphore is free. If the semaphore is free it is marked allocated, otherwise it gets the ID of the current owner. This is all done in one uninterruptible operation. In a single-processor system, the CMPXCHG instruction eliminates the need to switch to protection level 0 (to disable interrupts) before executing multiple instructions to test and modify a semaphore. For multiple processor systems, CMPXCHG can be combined with the LOCK prefix to perform the compare and exchange operation atomically. (See “Locked Atomic Operations” in Chapter 7 of the *Intel Architecture Software Developer’s Manual, Volume 3*, for more information on atomic operations.)

The CMPXCHG8B instruction also requires three operands: a 64-bit value in EDX:EAX, a 64-bit value in ECX:EBX, and a destination operand in memory. The instruction compares the 64-bit value in the EDX:EAX registers with the destination operand. If they are equal, the 64-bit value in the ECX:EBX register is stored in the destination operand. If the EDX:EAX register and the destination are not equal, the destination is loaded in the EDX:EAX register. The CMPXCHG8B instruction can be combined with the LOCK prefix to perform the operation atomically.

6.3.2. Stack Manipulation Instructions

The PUSH, POP, PUSHA (push all registers), and POPA (pop all registers) instructions move data to and from the stack. The PUSH instruction decrements the stack pointer (contained in the ESP register), then copies the source operand to the top of stack (see Figure 6-1). It operates on memory operands, immediate operands, and register operands (including segment registers). The PUSH instruction is commonly used to place parameters on the stack before calling a procedure. It can also be used to reserve space on the stack for temporary variables.

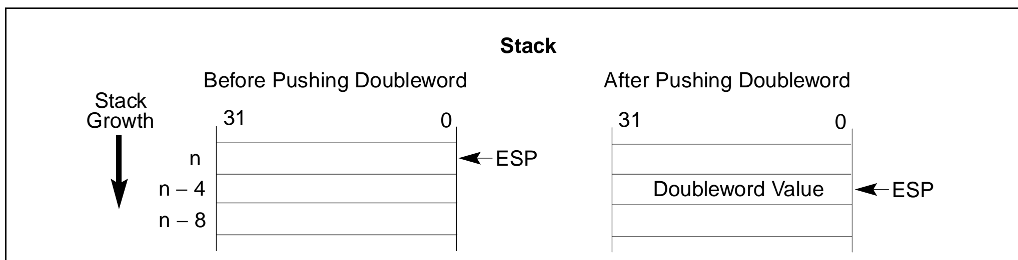


Figure 6-1. Operation of the PUSH Instruction

The PUSHA instruction saves the contents of the eight general-purpose registers on the stack (see Figure 6-2). This instruction simplifies procedure calls by reducing the number of instructions required to save the contents of the general-purpose registers. The registers are pushed on the stack in the following order: EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed, EBP, ESI, and EDI.

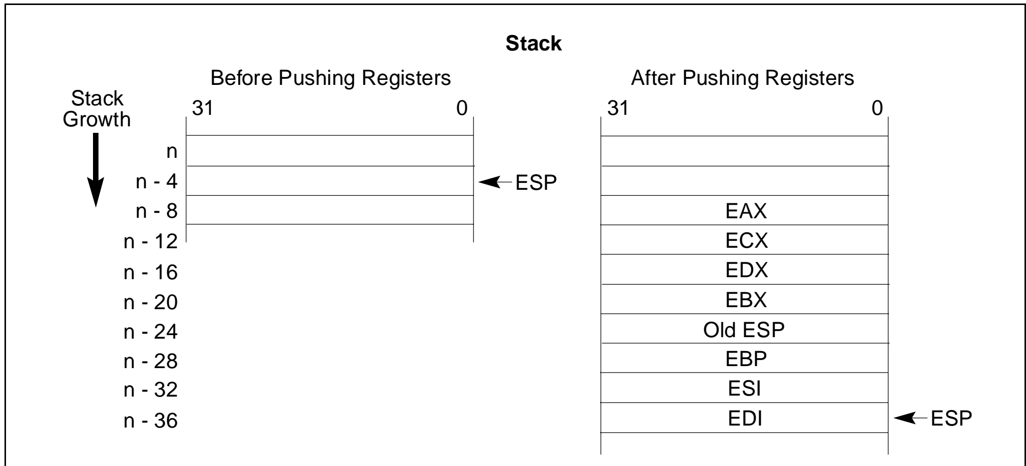


Figure 6-2. Operation of the PUSHA Instruction

The POP instruction copies the word or doubleword at the current top of stack (indicated by the ESP register) to the location specified with the destination operand, and then increments the ESP register to point to the new top of stack (see Figure 6-3). The destination operand may specify a general-purpose register, a segment register, or a memory location.

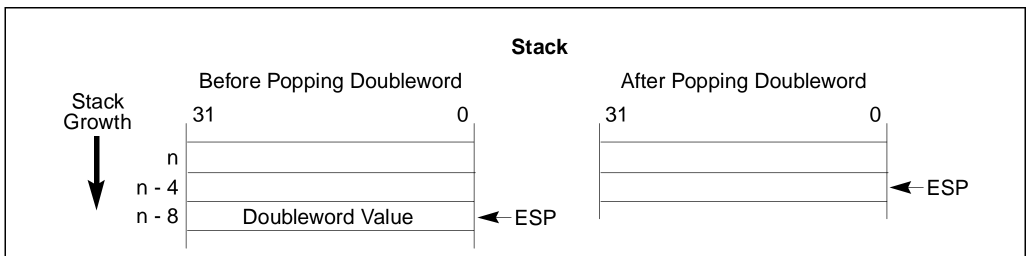


Figure 6-3. Operation of the POP Instruction

The POPA instruction reverses the effect of the PUSHA instruction. It pops the top eight words or doublewords from the top of the stack into the general-purpose registers, except for the ESP register (see Figure 6-4). If the operand-size attribute is 32, the doublewords on the stack are transferred to the registers in the following order: EDI, ESI, EBP, ignore doubleword, EBX, EDX, ECX, and EAX. The ESP register is restored by the action of popping the stack. If the operand-size attribute is 16, the words on the stack are transferred to the registers in the following order: DI, SI, BP, ignore word, BX, DX, CX, and AX.

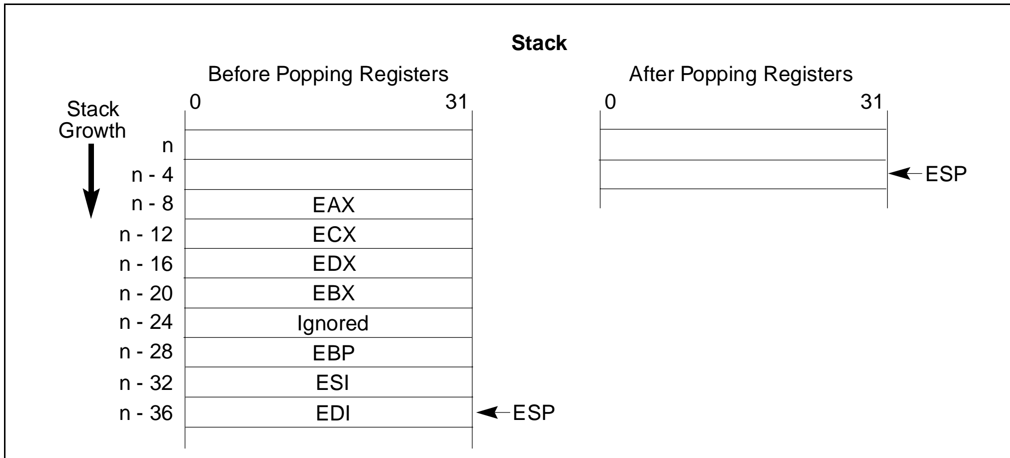


Figure 6-4. Operation of the POPA Instruction

6.3.2.1. TYPE CONVERSION INSTRUCTIONS

The type conversion instructions convert bytes into words, words into doublewords, and double-words into quadwords. These instructions are especially useful for converting integers to larger integer formats, because they perform sign extension (see Figure 6-5).

Two kinds of type conversion instructions are provided: simple conversion and move and convert.

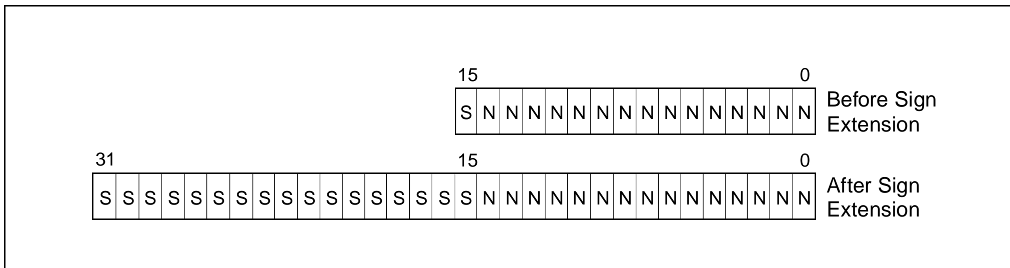


Figure 6-5. Sign Extension

6.3.2.2. SIMPLE CONVERSION

The CBW (convert byte to word), CWDE (convert word to doubleword extended), CWD (convert word to doubleword), and CDQ (convert doubleword to quadword) instructions perform sign extension to double the size of the source operand.

The CBW instruction copies the sign (bit 7) of the byte in the AL register into every bit position of the upper byte of the AX register. The CWDE instruction copies the sign (bit 15) of the word in the AX register into every bit position of the high word of the EAX register.

The CWD instruction copies the sign (bit 15) of the word in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the doubleword 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.

6.3.2.3. MOVE AND CONVERT

The MOVSX (move with sign extension) and MOVZX (move with zero extension) instructions move the source operand into a register then perform the sign extension.

The MOVSX instruction extends an 8-bit value to a 16-bit value or an 8- or 16-bit value to 32-bit value by sign extending the source operand, as shown in Figure 6-5. The MOVZX instruction extends an 8-bit value to a 16-bit value or an 8- or 16-bit value to 32-bit value by zero extending the source operand.

6.4. BINARY ARITHMETIC INSTRUCTIONS

The binary arithmetic instructions operate on 8-, 16-, and 32-bit numeric data encoded as signed or unsigned binary integers. Operations include the add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign (negate). The binary arithmetic instructions may also be used in algorithms that operate on decimal (BCD) values.

6.4.1. Addition and Subtraction Instructions

The ADD (add integers), ADC (add integers with carry), SUB (subtract integers), and SBB (subtract integers with borrow) instructions perform addition and subtraction operations on signed or unsigned integer operands.

The ADD instruction computes the sum of two integer operands.

The ADC instruction computes the sum of two integer operands, plus 1 if the CF flag is set. This instruction is used to propagate a carry when adding numbers in stages.

The SUB instruction computes the difference of two integer operands.

The SBB instruction computes the difference of two integer operands, minus 1 if the CF flag is set. This instruction is used to propagate a borrow when subtracting numbers in stages.

6.4.2. Increment and Decrement Instructions

The INC (increment) and DEC (decrement) instructions add 1 to or subtract 1 from an unsigned integer operand, respectively. A primary use of these instructions is for implementing counters.

6.4.3. Comparison and Sign Change Instruction

The CMP (compare) instruction computes the difference between two integer operands and updates the OF, SF, ZF, AF, PF, and CF flags according to the result. The source operands are not modified, nor is the result saved. The CMP instruction is commonly used in conjunction with a J*cc* (jump) or SET*cc* (byte set on condition) instruction, with the latter instructions performing an action based on the result of a CMP instruction.

The NEG (negate) instruction subtracts a signed integer operand from zero. The effect of the NEG instruction is to change the sign of a two's complement operand while keeping its magnitude.

6.4.4. Multiplication and Divide Instructions

The processor provides two multiply instructions, MUL (unsigned multiply) and IMUL signed multiply), and two divide instructions, DIV (unsigned divide) and IDIV (signed divide).

The MUL instruction multiplies two unsigned integer operands. The result is computed to twice the size of the source operands (for example, if word operands are being multiplied, the result is a doubleword).

The IMUL instruction multiplies two signed integer operands. The result is computed to twice the size of the source operands; however, in some cases the result is truncated to the size of the source operands (see “IMUL—Signed Multiply” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 2*).

The DIV instruction divides one unsigned operand by another unsigned operand and returns a quotient and a remainder.

The IDIV instruction is identical to the DIV instruction, except that IDIV performs a signed division.

6.5. DECIMAL ARITHMETIC INSTRUCTIONS

Decimal arithmetic can be performed by combining the binary arithmetic instructions ADD, SUB, MUL, and DIV (discussed in Section 6.4., “Binary Arithmetic Instructions”) with the decimal arithmetic instructions. The decimal arithmetic instructions are provided to carry out the following operations:

- To adjust the results of a previous binary arithmetic operation to produce a valid BCD result.
- To adjust the operands of a subsequent binary arithmetic operation so that the operation will produce a valid BCD result.

These instructions operate only on both packed and unpacked BCD values.

6.5.1. Packed BCD Adjustment Instructions

The DAA (decimal adjust after addition) and DAS (decimal adjust after subtraction) instructions adjust the results of operations performed on packed BCD integers (see Section 5.2.3., "BCD Integers"). Adding two packed BCD values requires two instructions: an ADD instruction followed by a DAA instruction. The ADD instruction adds (binary addition) the two values and stores the result in the AL register. The DAA instruction then adjusts the value in the AL register to obtain a valid, 2-digit, packed BCD value and sets the CF flag if a decimal carry occurred as the result of the addition.

Likewise, subtracting one packed BCD value from another requires a SUB instruction followed by a DAS instruction. The SUB instruction subtracts (binary subtraction) one BCD value from another and stores the result in the AL register. The DAS instruction then adjusts the value in the AL register to obtain a valid, 2-digit, packed BCD value and sets the CF flag if a decimal borrow occurred as the result of the subtraction.

6.5.2. Unpacked BCD Adjustment Instructions

The AAA (ASCII adjust after addition), AAS (ASCII adjust after subtraction), AAM (ASCII adjust after multiplication), and AAD (ASCII adjust before division) instructions adjust the results of arithmetic operations performed in unpacked BCD values (see Section 5.2.3., "BCD Integers"). All these instructions assume that the value to be adjusted is stored in the AL register or, in one instance, the AL and AH registers.

The AAA instruction adjusts the contents of the AL register following the addition of two unpacked BCD values. It converts the binary value in the AL register into a decimal value and stores the result in the AL register in unpacked BCD format (the decimal number is stored in the lower 4 bits of the register and the upper 4 bits are cleared). If a decimal carry occurred as a result of the addition, the CF flag is set and the contents of the AH register are incremented by 1.

The AAS instruction adjusts the contents of the AL register following the subtraction of two unpacked BCD values. Here again, a binary value is converted into an unpacked BCD value. If a borrow was required to complete the decimal subtract, the CF flag is set and the contents of the AH register are decremented by 1.

The AAM instruction adjusts the contents of the AL register following a multiplication of two unpacked BCD values. It converts the binary value in the AL register into a decimal value and stores the least significant digit of the result in the AL register (in unpacked BCD format) and the most significant digit, if there is one, in the AH register (also in unpacked BCD format).

The AAD instruction adjusts a two-digit BCD value so that when the value is divided with the DIV instruction, a valid unpacked BCD result is obtained. The instruction converts the BCD value in registers AH (most significant digit) and AL (least significant digit) into a binary value and stores the result in register AL. When the value in AL is divided by an unpacked BCD value, the quotient and remainder will be automatically encoded in unpacked BCD format.

6.6. LOGICAL INSTRUCTIONS

The logical instructions AND, OR, XOR (exclusive or), and NOT perform the standard Boolean operations for which they are named. The AND, OR, and XOR instructions require two operands; the NOT instruction operates on a single operand.

6.7. SHIFT AND ROTATE INSTRUCTIONS

The shift and rotate instructions rearrange the bits within an operand. These instructions fall into the following classes:

- Shift.
- Double shift.
- Rotate.

6.7.1. Shift Instructions

The SAL (shift arithmetic left), SHL (shift logical left), SAR (shift arithmetic right), SHR (shift logical right) instructions perform an arithmetic or logical shift of the bits in a byte, word, or doubleword.

The SAL and SHL instructions perform the same operation (see Figure 6-6). They shift the source operand left by from 1 to 31 bit positions. Empty bit positions are cleared. The CF flag is loaded with the last bit shifted out of the operand.

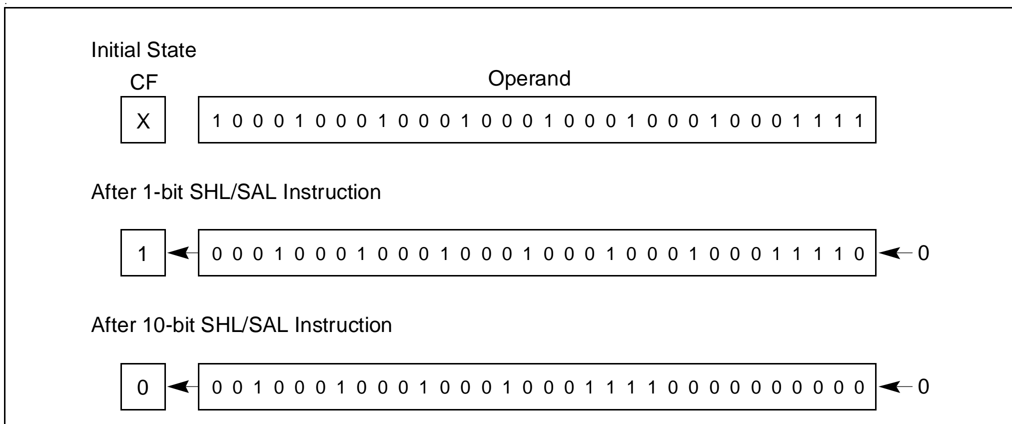


Figure 6-6. SHL/SAL Instruction Operation

The SHR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 6-7). As with the SHL/SAL instruction, the empty bit positions are cleared and the CF flag is loaded with the last bit shifted out of the operand.

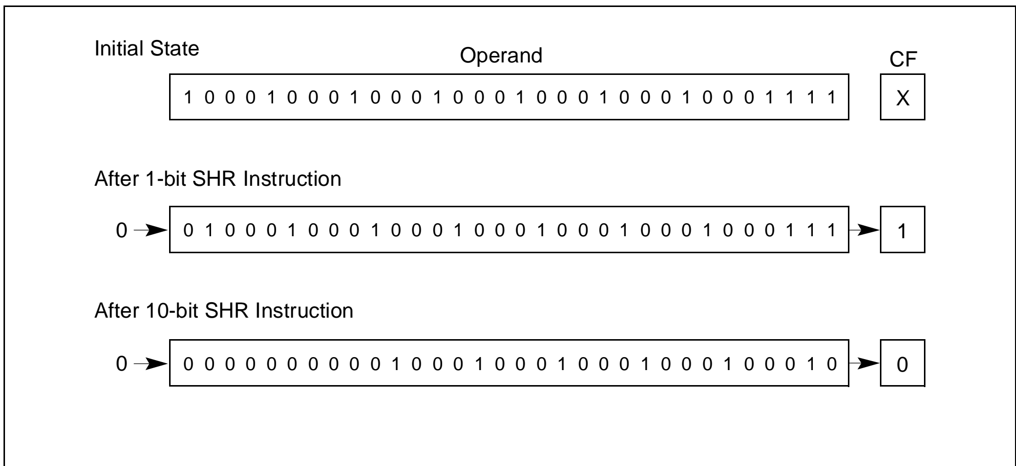


Figure 6-7. SHR Instruction Operation

The SAR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 6-8). This instruction differs from the SHR instruction in that it preserves the sign of the source operand by clearing empty bit positions if the operand is positive or setting the empty bits if the operand is negative. Again, the CF flag is loaded with the last bit shifted out of the operand.

The SAR and SHR instructions can also be used to perform division by powers of 2 (see "SAL/SAR/SHL/SHR—Shift Instructions" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*).

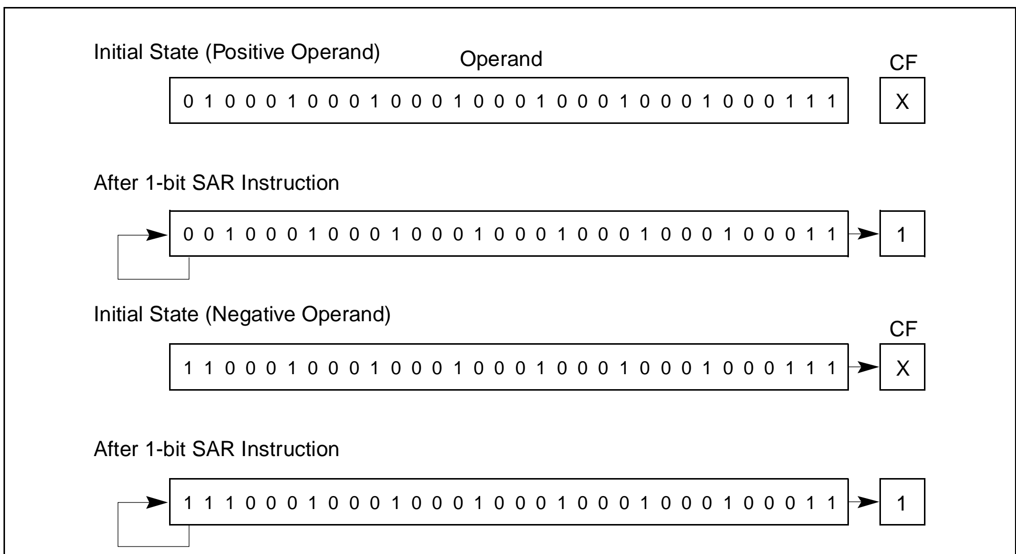


Figure 6-8. SAR Instruction Operation

6.7.2. Double-Shift Instructions

The SHLD (shift left double) and SHRD (shift right double) instructions shift a specified number of bits from one operand to another (see Figure 6-9). They are provided to facilitate operations on unaligned bit strings. They can also be used to implement a variety of bit string move operations.

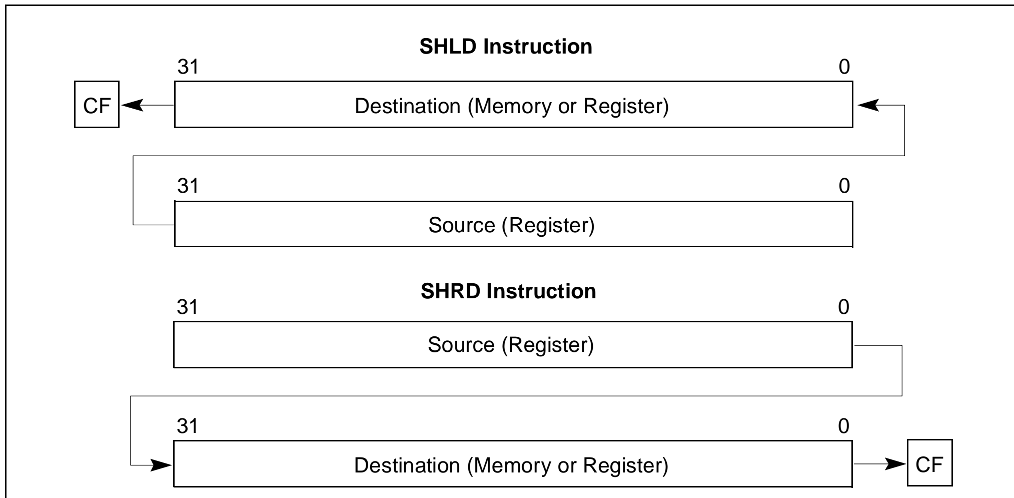


Figure 6-9. SHLD and SHRD Instruction Operations

The SHLD instruction shifts the bits in the destination operand to the left and fills the empty bit positions (in the destination operand) with bits shifted out of the source operand. The destination and source operands must be the same length (either words or doublewords). The shift count can range from 0 to 31 bits. The result of this shift operation is stored in the destination operand, and the source operand is not modified. The CF flag is loaded with the last bit shifted out of the destination operand.

The SHRD instruction operates the same as the SHLD instruction except bits are shifted to the left in the destination operand, with the empty bit positions filled with bits shifted out of the source operand.

6.7.3. Rotate Instructions

The ROL (rotate left), ROR (rotate right), RCL (rotate through carry left) and RCR (rotate through carry right) instructions rotate the bits in the destination operand out of one end and back through the other end (see Figure 6-10). Unlike a shift, no bits are lost during a rotation. The rotate count can range from 0 to 31.

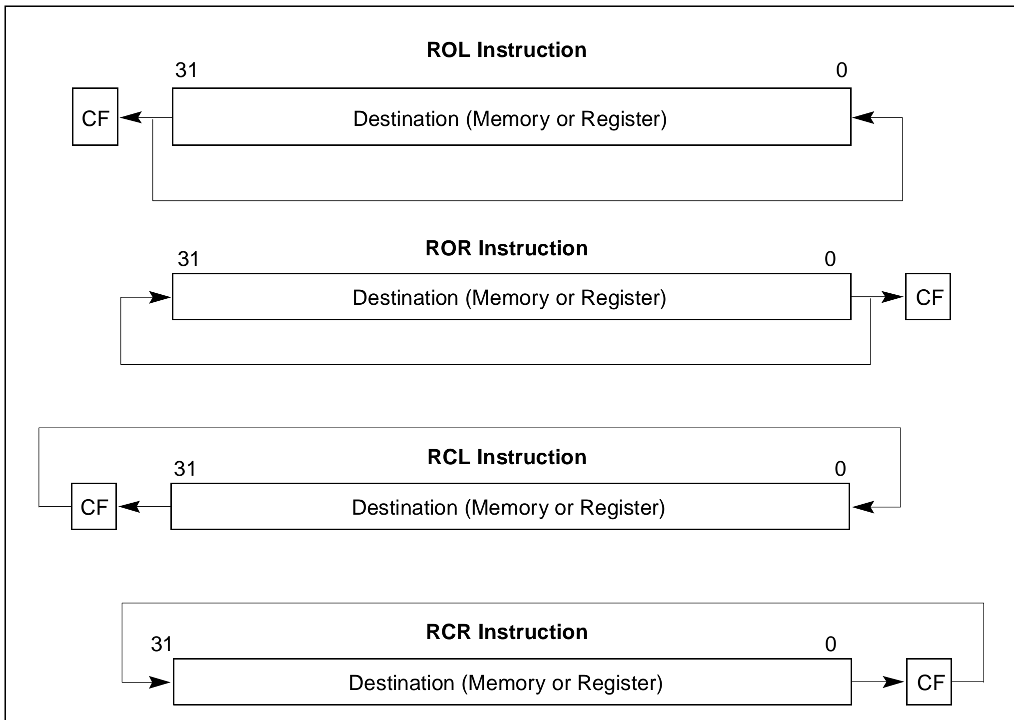


Figure 6-10. ROL, ROR, RCL, and RCR Instruction Operations

The ROL instruction rotates the bits in the operand to the left (toward more significant bit locations). The ROR instruction rotates the operand right (toward less significant bit locations).

The RCL instruction rotates the bits in the operand to the left, through the CF flag). This instruction treats the CF flag as a one-bit extension on the upper end of the operand. Each bit which exits from the most significant bit location of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the least significant bit location of the operand.

The RCR instruction rotates the bits in the operand to the right through the CF flag.

For all the rotate instructions, the CF flag always contains the value of the last bit rotated out of the operand, even if the instruction does not use the CF flag as an extension of the operand. The value of this flag can then be tested by a conditional jump instruction (JC or JNC).

6.8. BIT AND BYTE INSTRUCTIONS

The bit and byte instructions operate on bit or byte strings. They are divided into four groups:

- Bit test and modify instructions.
- Bit scan instructions.
- Byte set on condition.
- Test.

6.8.1. Bit Test and Modify Instructions

The bit test and modify instructions (see Table 6-3) operate on a single bit, which can be in an operand. The location of the bit is specified as an offset from the least significant bit of the operand. When the processor identifies the bit to be tested and modified, it first loads the CF flag with the current value of the bit. Then it assigns a new value to the selected bit, as determined by the modify operation for the instruction.

Table 6-3. Bit Test and Modify Instructions

Instruction	Effect on CF Flag	Effect on Selected Bit
BT (Bit Test)	CF flag ← Selected Bit	No effect
BTS (Bit Test and Set)	CF flag ← Selected Bit	Selected Bit ← 1
BTR (Bit Test and Reset)	CF flag ← Selected Bit	Selected Bit ← 0
BTC (Bit Test and Complement)	CF flag ← Selected Bit	Selected Bit ← NOT (Selected Bit)

6.8.2. Bit Scan Instructions

The BSF (bit scan forward) and BSR (bit scan reverse) instructions scan a bit string in a source operand for a set bit and store the bit index of the first set bit found in a destination register. The bit index is the offset from the least significant bit (bit 0) in the bit string to the first set bit. The BSF instruction scans the source operand low-to-high (from bit 0 of the source operand toward the most significant bit); the BSR instruction scans high-to-low (from the most significant bit toward the least significant bit).

6.8.3. Byte Set On Condition Instructions

The SET*cc* (set byte on condition) instructions set a destination-operand byte to 0 or 1, depending on the state of selected status flags (CF, OF, SF, ZF, and PF) in the EFLAGS register. The suffix (*cc*) added to the SET mnemonic determines the condition being tested for. For example, the SETO instruction tests for overflow. If the OF flag is set, the destination byte is set to 1; if OF is clear, the destination byte is cleared to 0. Appendix B, *EFLAGS Condition Codes* lists the conditions it is possible to test for with this instruction.

6.8.4. Test Instruction

The TEST instruction performs a logical AND of two operands and sets the SF, ZF, and PF flags according to the results. The flags can then be tested by the conditional jump or loop instructions or the SET*cc* instructions. The TEST instruction differs from the AND instruction in that it does not alter either of the operands.

6.9. CONTROL TRANSFER INSTRUCTIONS

The processor provides both conditional and unconditional control transfer instructions to direct the flow of program execution. Conditional transfers are taken only for specified states of the status flags in the EFLAGS register. Unconditional control transfers are always executed.

6.9.1. Unconditional Transfer Instructions

The JMP, CALL, RET, INT, and IRET instructions transfer program control to another location (destination address) in the instruction stream. The destination can be within the same code segment (near transfer) or in a different code segment (far transfer).

6.9.1.1. JUMP INSTRUCTION

The JMP (jump) instruction unconditionally transfers program control to a destination instruction. The transfer is one-way; that is, a return address is not saved. A destination operand specifies the address (the instruction pointer) of the destination instruction. The address can be a **relative address** or an **absolute address**.

A relative address is a displacement (offset) with respect to the address in the EIP register. The destination address (a near pointer) is formed by adding the displacement to the address in the EIP register. The displacement is specified with a signed integer, allowing jumps either forward or backward in the instruction stream.

An absolute address is a offset from address 0 of a segment. It can be specified in either of the following ways:

- **An address in a general-purpose register.** This address is treated as a near pointer, which is copied into the EIP register. Program execution then continues at the new address within the current code segment.
- **An address specified using the standard addressing modes of the processor.** Here, the address can be a near pointer or a far pointer. If the address is for a near pointer, the address is translated into an offset and copied into the EIP register. If the address is for a far pointer, the address is translated into a segment selector (which is copied into the CS register) and an offset (which is copied into the EIP register).

In protected mode, the JMP instruction also allows jumps to a call gate, a task gate, and a task-state segment.

6.9.1.2. CALL AND RETURN INSTRUCTIONS

The CALL (call procedure) and RET (return from procedure) instructions allow a jump from one procedure (or subroutine) to another and a subsequent jump back (return) to the calling procedure.

The CALL instruction transfers program control from the current (or calling procedure) to another procedure (the called procedure). To allow a subsequent return to the calling procedure, the CALL instruction saves the current contents of the EIP register on the stack before jumping to the called procedure. The EIP register (prior to transferring program control) contains the address of the instruction following the CALL instruction. When this address is pushed on the stack, it is referred to as the **return instruction pointer** or **return address**.

The address of the called procedure (the address of the first instruction in the procedure being jumped to) is specified in a CALL instruction the same way as it is in a JMP instruction (see Section 6.9.1.1., "Jump Instruction"). The address can be specified as a relative address or an absolute address. If an absolute address is specified, it can be either a near or a far pointer.

The RET instruction transfers program control from the procedure currently being executed (the called procedure) back to the procedure that called it (the calling procedure). Transfer of control is accomplished by copying the return instruction pointer from the stack into the EIP register. Program execution then continues with the instruction pointed to by the EIP register.

The RET instruction has an optional operand, the value of which is added to the contents of the ESP register as part of the return operation. This operand allows the stack pointer to be incremented to remove parameters from the stack that were pushed on the stack by the calling procedure.

See Section 4.3., "Calling Procedures Using CALL and RET", for more information on the mechanics of making procedure calls with the CALL and RET instructions.

6.9.1.3. RETURN FROM INTERRUPT INSTRUCTION

When the processor services an interrupt, it performs an implicit call to an interrupt-handling procedure. The IRET (return from interrupt) instruction returns program control from an interrupt handler to the interrupted procedure (that is, the procedure that was executing when the interrupt occurred). The IRET instruction performs a similar operation to the RET instruction (see Section 6.9.1.2., "Call and Return Instructions") except that it also restores the EFLAGS register from the stack. The contents of the EFLAGS register are automatically stored on the stack along with the return instruction pointer when the processor services an interrupt.

6.9.2. Conditional Transfer Instructions

The conditional transfer instructions execute jumps or loops that transfer program control to another instruction in the instruction stream if specified conditions are met. The conditions for control transfer are specified with a set of condition codes that define various states of the status flags (CF, ZF, OF, PF, and SF) in the EFLAGS register.

6.9.2.1. CONDITIONAL JUMP INSTRUCTIONS

The J*cc* (conditional) jump instructions transfer program control to a destination instruction if the conditions specified with the condition code (*cc*) associated with the instruction are satisfied (see Table 6-4). If the condition is not satisfied, execution continues with the instruction following the J*cc* instruction. As with the JMP instruction, the transfer is one-way; that is, a return address is not saved.

Table 6-4. Conditional Jump Instructions

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF)=0	Above/not below or equal
JAE/JNB	CF=0	Above or equal/not below
JB/JNAE	CF=1	Below/not above or equal
JBE/JNA	(CF or ZF)=1	Below or equal/not above
JC	CF=1	Carry
JE/JZ	ZF=1	Equal/zero
JNC	CF=0	Not carry
JNE/JNZ	ZF=0	Not equal/not zero
JNP/JPO	PF=0	Not parity/parity odd
JP/JPE	PF=1	Parity/parity even
JCXZ	CX=0	Register CX is zero
JECXZ	ECX=0	Register ECX is zero
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) =0	Greater/not less or equal
JGE/JNL	(SF xor OF)=0	Greater or equal/not less
JL/JNGE	(SF xor OF)=1	Less/not greater or equal
JLE/JNG	((SF xor OF) or ZF)=1	Less or equal/not greater
JNO	OF=0	Not overflow
JNS	SF=0	Not sign (non-negative)
JO	OF=1	Overflow
JS	SF=1	Sign (negative)

The destination operand specifies a relative address (a signed offset with respect to the address in the EIP register) that points to an instruction in the current code segment. The J*cc* instructions do not support far transfers; however, far transfers can be accomplished with a combination of a J*cc* and a JMP instruction (see "J*cc*—Jump if Condition Is Met" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*).

Table 6-4 shows the mnemonics for the J*cc* instructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letter "J" to form the mnemonic for a J*cc* instruction. The instructions are divided into two groups: unsigned and signed conditional jumps. These groups correspond to the results of operations performed on unsigned and signed integers, respectively. Those instructions listed as pairs (for example, JA/JNBE) are alternate names for the same instruction. The assembler provides these alternate names to make it easier to read program listings.

The JCXZ and JECXZ instructions test the CX and ECX registers, respectively, instead of one or more status flags. See Section 6.9.2.3., "Jump If Zero Instructions" for more information about these instructions.

6.9.2.2. LOOP INSTRUCTIONS

The LOOP, LOOPE (loop while equal), LOOPZ (loop while zero), LOOPNE (loop while not equal), and LOOPNZ (loop while not zero) instructions are conditional jump instructions that use the value of the ECX register as a count for the number of times to execute a loop. All the loop instructions decrement the count in the ECX register each time they are executed and terminate a loop when zero is reached. The LOOPE, LOOPZ, LOOPNE, and LOOPNZ instructions also accept the ZF flag as a condition for terminating the loop before the count reaches zero.

The LOOP instruction decrements the contents of the ECX register (or the CX register, if the address-size attribute is 16), then tests the register for the loop-termination condition. If the count in the ECX register is non-zero, program control is transferred to the instruction address specified by the destination operand. The destination operand is a relative address (that is, an offset relative to the contents of the EIP register), and it generally points to the first instruction in the block of code that is to be executed in the loop. When the count in the ECX register reaches zero, program control is transferred to the instruction immediately following the LOOP instruction, which terminates the loop. If the count in the ECX register is zero when the LOOP instruction is first executed, the register is pre-decremented to FFFFFFFFH, causing the loop to be executed 2^{32} times.

The LOOPE and LOOPZ instructions perform the same operation (they are mnemonics for the same instruction). These instructions operate the same as the LOOP instruction, except that they also test the ZF flag. If the count in the ECX register is not zero and the ZF flag is set, program control is transferred to the destination operand. When the count reaches zero or the ZF flag is clear, the loop is terminated by transferring program control to the instruction immediately following the LOOPE/LOOPZ instruction.

The LOOPNE and LOOPNZ instructions (mnemonics for the same instruction) operate the same as the LOOPE/LOOPPZ instructions, except that they terminate the loop if the ZF flag is set.

6.9.2.3. JUMP IF ZERO INSTRUCTIONS

The JECXZ (jump if ECX zero) instruction jumps to the location specified in the destination operand if the ECX register contains the value zero. This instruction can be used in combination with a loop instruction (LOOP, LOOPE, LOOPZ, LOOPNE, or LOOPNZ) to test the ECX register prior to beginning a loop. As described in Section 6.9.2.2., "Loop Instructions", the loop

instructions decrement the contents of the ECX register before testing for zero. If the value in the ECX register is zero initially, it will be decremented to FFFFFFFFH on the first loop instruction, causing the loop to be executed 2^{32} times. To prevent this problem, a JECXZ instruction can be inserted at the beginning of the code block for the loop, causing a jump out the loop if the EAX register count is initially zero. When used with repeated string scan and compare instructions, the JECXZ instruction can determine whether the loop terminated because the count reached zero or because the scan or compare conditions were satisfied.

The JCXZ (jump if CX is zero) instruction operates the same as the JECXZ instruction when the 16-bit address-size attribute is used. Here, the CX register is tested for zero.

6.9.3. Software Interrupts

The INT *n* (software interrupt), INTO (interrupt on overflow), and BOUND (detect value out of range) instructions allow a program to explicitly raise a specified interrupt or exception, which in turn causes the handler routine for the interrupt or exception to be called.

The INT *n* instruction can raise any of the processor's interrupts or exceptions by encoding the vector number or the interrupt or exception in the instruction. This instruction can be used to support software generated interrupts or to test the operation of interrupt and exception handlers. The IRET instruction (see Section 6.9.1.3., "Return From Interrupt Instruction") allows returns from interrupt handling routines.

The INTO instruction raises the overflow exception, if the OF flag is set. If the flag is clear, execution continues without raising the exception. This instruction allows software to access the overflow exception handler explicitly to check for overflow conditions.

The BOUND instruction compares a signed value against upper and lower bounds, and raises the "BOUND range exceeded" exception if the value is less than the lower bound or greater than the upper bound. This instruction is useful for operations such as checking an array index to make sure it falls within the range defined for the array.

6.10. STRING OPERATIONS

The MOVS (Move String), CMPS (Compare string), SCAS (Scan string), LODS (Load string), and STOS (Store string) instructions permit large data structures, such as alphanumeric character strings, to be moved and examined in memory. These instructions operate on individual elements in a string, which can be a byte, word, or doubleword. The string elements to be operated on are identified with the ESI (source string element) and EDI (destination string element) registers. Both of these registers contain absolute addresses (offsets into a segment) that point to a string element.

By default, the ESI register addresses the segment identified with the DS segment register. A segment-override prefix allows the ESI register to be associated with the CS, SS, ES, FS, or GS segment register. The EDI register addresses the segment identified with the ES segment register; no segment override is allowed for the EDI register. The use of two different segment registers in the string instructions permits operations to be performed on strings located in different segments. Or by associating the ESI register with the ES segment register, both the

source and destination strings can be located in the same segment. (This latter condition can also be achieved by loading the DS and ES segment registers with the same segment selector and allowing the ESI register to default to the DS register.)

The MOVS instruction moves the string element addressed by the ESI register to the location addressed by the EDI register. The assembler recognizes three “short forms” of this instruction, which specify the size of the string to be moved: MOVSB (move byte string), MOVSW (move word string), and MOVSD (move doubleword string).

The CMPS instruction subtracts the destination string element from the source string element and updates the status flags (CF, ZF, OF, SF, PF, and AF) in the EFLAGS register according to the results. Neither string element is written back to memory. The assembler recognizes three “short forms” of the CMPS instruction: CMPSB (compare byte strings), CMPSW (compare word strings), and CMPSD (compare doubleword strings).

The SCAS instruction subtracts the destination string element from the contents of the EAX, AX, or AL register (depending on operand length) and updates the status flags according to the results. The string element and register contents are not modified. The following “short forms” of the SCAS instruction specifies the operand length: SCASB (scan byte string), SCASW (scan word string), and SCASD (scan doubleword string).

The LODS instruction loads the source string element identified by the ESI register into the EAX register (for a doubleword string), the AX register (for a word string), or the AL register (for a byte string). The “short forms” for this instruction are LODSB (load byte string), LODSW (load word string), and LODSD (load doubleword string). This instruction is usually used in a loop, where other instructions process each element of the string after they are loaded into the target register.

The STOS instruction stores the source string element from the EAX (doubleword string), AX (word string), or AL (byte string) register into the memory location identified with the EDI register. The “short forms” for this instruction are STOSB (store byte string), STOSW (store word string), and STOSD (store doubleword string). This instruction is also normally used in a loop. Here a string is commonly loaded into the register with a LODS instruction, operated on by other instructions, and then stored again in memory with a STOS instruction.

The I/O instructions (see Section 6.11., “I/O Instructions”) also perform operations on strings in memory.

6.10.1. Repeating String Operations

The string instructions described in Section 6.10., “String Operations” perform one iteration of a string operation. To operate strings longer than a doubleword, the string instructions can be combined with a repeat prefix (REP) to create a repeating instruction or be placed in a loop.

When used in string instructions, the ESI and EDI registers are automatically incremented or decremented after each iteration of an instruction to point to the next element (byte, word, or doubleword) in the string. String operations can thus begin at higher addresses and work toward lower ones, or they can begin at lower addresses and work toward higher ones. The DF flag in

the EFLAGS register controls whether the registers are incremented (DF=0) or decremented (DF=1). The STD and CLD instructions set and clear this flag, respectively.

The following repeat prefixes can be used in conjunction with a count in the ECX register to cause a string instruction to repeat:

- REP—Repeat while the ECX register not zero.
- REPE/REPZ—Repeat while the ECX register not zero and the ZF flag is set.
- REPNE/REPNZ—Repeat while the ECX register not zero and the ZF flag is clear.

When a string instruction has a repeat prefix, the operation executes until one of the termination conditions specified by the prefix is satisfied. The REPE/REPZ and REPNE/REPNZ prefixes are used only with the CMPS and SCAS instructions. Also, note that a A REP STOS instruction is the fastest way to initialize a large block of memory.

6.11. I/O INSTRUCTIONS

The IN (input from port to register), INS (input from port to string), OUT (output from register to port), and OUTS (output string to port) instructions move data between the processor's I/O ports and either a register or memory.

The register I/O instructions (IN and OUT) move data between an I/O port and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register. The I/O port being read or written to is specified with an immediate operand or an address in the DX register.

The block I/O instructions (INS and OUTS) instructions move blocks of data (strings) between an I/O port and memory. These instructions operate similar to the string instructions (see Section 6.10., "String Operations"). The ESI and EDI registers are used to specify string elements in memory and the repeat prefixes (REP) are used to repeat the instructions to implement block moves. The assembler recognizes the following alternate mnemonics for these instructions: INSB (input byte), INSW (input word), and INSD (input doubleword), and OUTB (output byte), OUTW (output word), and OUTD (output doubleword).

The INS and OUTS instructions use an address in the DX register to specify the I/O port to be read or written to.

6.12. ENTER AND LEAVE INSTRUCTIONS

The ENTER and LEAVE instructions provide machine-language support for procedure calls in block-structured languages, such as C and Pascal. These instructions and the call and return mechanism that they support are described in detail in Section 4.5., "Procedure Calls for Block-Structured Languages".

6.13. EFLAGS INSTRUCTIONS

The EFLAGS instructions allow the state of selected flags in the EFLAGS register to be read or modified.

6.13.1. Carry and Direction Flag Instructions

The STC (set carry flag), CLC (clear carry flag), and CMC (complement carry flag) instructions allow the CF flags in the EFLAGS register to be modified directly. They are typically used to initialize the CF flag to a known state before an instruction that uses the flag in an operation is executed. They are also used in conjunction with the rotate-with-carry instructions (RCL and RCR).

The STD (set direction flag) and CLD (clear direction flag) instructions allow the DF flag in the EFLAGS register to be modified directly. The DF flag determines the direction in which index registers ESI and EDI are stepped when executing string processing instructions. If the DF flag is clear, the index registers are incremented after each iteration of a string instruction; if the DF flag is set, the registers are decremented.

6.13.2. Interrupt Flag Instructions

The STI (set interrupt flag) and CTI (clear interrupt flag) instructions allow the interrupt IF flag in the EFLAGS register to be modified directly. The IF flag controls the servicing of hardware-generated interrupts (those received at the processor's INTR pin). If the IF flag is set, the processor services hardware interrupts; if the IF flag is clear, hardware interrupts are masked.

6.13.3. EFLAGS Transfer Instructions

The EFLAGS transfer instructions allow groups of flags in the EFLAGS register to be copied to a register or memory or be loaded from a register or memory.

The LAHF (load AH from flags) and SAHF (store AH into flags) instructions operate on five of the EFLAGS status flags (SF, ZF, AF, PF, and CF). The LAHF instruction copies the status flags to bits 7, 6, 4, 2, and 0 of the AH register, respectively. The contents of the remaining bits in the register (bits 5, 3, and 1) are undefined, and the contents of the EFLAGS register remain unchanged. The SAHF instruction copies bits 7, 6, 4, 2, and 0 from the AH register into the SF, ZF, AF, PF, and CF flags, respectively in the EFLAGS register.

The PUSHF (push flags), PUSHFD (push flags double), POPF (pop flags), and POPFD (pop flags double) instructions copy the flags in the EFLAGS register to and from the stack. The PUSHF instruction pushes the lower word of the EFLAGS register onto the stack (see Figure 6-11). The PUSHFD instruction pushes the entire EFLAGS register onto the stack (with the RF and VM flags read as clear).

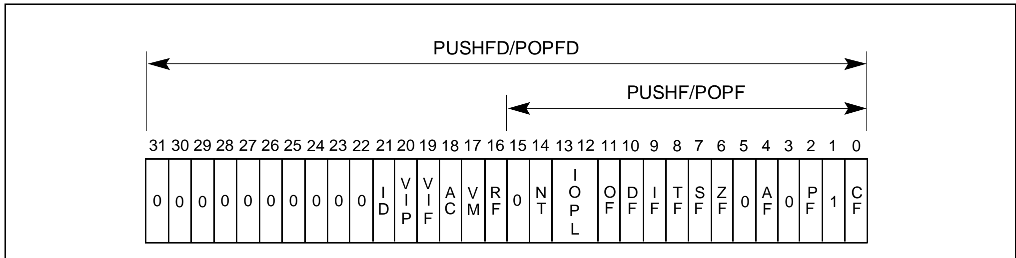


Figure 6-11. Flags Affected by the PUSHF, POPF, PUSHFD, and POPFD instructions

The POPF instruction pops a word from the stack into the EFLAGS register. Only bits 11, 10, 8, 7, 6, 4, 2, and 0 of the EFLAGS register are affected with all uses of this instruction. If the current privilege level (CPL) of the current code segment is 0 (most privileged), the IOPL bits (bits 13 and 12) also are affected. If the I/O privilege level (IOPL) is greater than or equal to the CPL, numerically, the IF flag (bit 9) also is affected.

The POPFD instruction pops a doubleword into the EFLAGS register. This instruction can change the state of the AC bit (bit 18) and the ID bit (bit 21), as well as the bits affected by a POPF instruction. The restrictions for changing the IOPL bits and the IF flag that were given for the POPF instruction also apply to the POPFD instruction.

6.13.4. Interrupt Flag Instructions

The CLI (clear interrupt flag) and STI (set interrupt flag) instructions clear and set the interrupt flag (IF) in the EFLAGS register, respectively. Clearing the IF flag causes external interrupts to be ignored. The ability to execute these instructions depends on the operating mode of the processor and the current privilege level (CPL) of the program or task attempting to execute these instructions.

6.14. SEGMENT REGISTER INSTRUCTIONS

The processor provides a variety of instructions that address the segment registers of the processor directly. These instructions are only used when an operating system or executive is using the segmented or the real-address mode memory model.

6.14.1. Segment-Register Load and Store Instructions

The MOV instruction (introduced in Section 6.3.1., "General-Purpose Data Movement Instructions") and the PUSH and POP instructions (introduced in Section 6.3.2., "Stack Manipulation Instructions") can transfer 16-bit segment selectors to and from segment registers (DS, ES, FS, GS, and SS). The transfers are always made to or from a segment register and a general-purpose register or memory. Transfers between segment registers are not supported.

The POP and MOV instructions cannot place a value in the CS register. Only the far control-transfer versions of the JMP, CALL, and RET instructions (see Section 6.14.2., "Far Control Transfer Instructions") affect the CS register directly.

6.14.2. Far Control Transfer Instructions

The JMP and CALL instructions (see Section 6.9., "Control Transfer Instructions") both accept a far pointer as a source operand to transfer program control to a segment other than the segment currently being pointed to by the CS register. When a far call is made with the CALL instruction, the current values of the EIP and CS registers are both pushed on the stack.

The RET instruction (see Section 6.9.1.2., "Call and Return Instructions") can be used to execute a far return. Here, program control is transferred from a code segment that contains a called procedure back to the code segment that contained the calling procedure. The RET instruction restores the values of the CS and EIP registers for the calling procedure from the stack.

6.14.3. Software Interrupt Instructions

The software interrupt instructions INT, INTO, BOUND, and IRET (see Section 6.9.3., "Software Interrupts") can also call and return from interrupt and exception handler procedures that are located in a code segment other than the current code segment. With these instructions, however, the switching of code segments is handled transparently from the application program.

6.14.4. Load Far Pointer Instructions

The load far pointer instructions LDS (load far pointer using DS), LES (load far pointer using ES), LFS (load far pointer using FS), LGS (load far pointer using GS), and LSS (load far pointer using SS) load a far pointer from memory into a segment register and a general-purpose general register. The segment selector part of the far pointer is loaded into the selected segment register and the offset is loaded into the selected general-purpose register.

6.15. MISCELLANEOUS INSTRUCTIONS

The following instructions perform miscellaneous operations that are of interest to applications programmers.

6.15.1. Address Computation Instruction

The LEA (load effective address) instruction computes the effective address in memory (offset within a segment) of a source operand and places it in a general-purpose register. This instruction can interpret any of the Pentium Pro processor's addressing modes and can perform any indexing or scaling that may be needed. It is especially useful for initializing the ESI or EDI

registers before the execution of string instructions or for initializing the EBX register before an XLAT instruction.

6.15.2. Table Lookup Instructions

The XLAT and XLATB (table lookup) instructions replace the contents of the AL register with a byte read from a translation table in memory. The initial value in the AL register is interpreted as an unsigned index into the translation table. This index is added to the contents of the EBX register (which contains the base address of the table) to calculate the address of the table entry. These instructions are used for applications such as converting character codes from one alphabet into another (for example, an ASCII code could be used to look up its EBCDIC equivalent in a table).

6.15.3. Processor Identification Instruction

The CPUID (processor identification) instruction provides information about the processor on which the instruction is executed. To obtain processor information, a value of from 0 to 2 is loaded in the EAX register and then the CPUID instruction is executed. The resulting processor information is placed in the EAX, EBX, ECX, and EDX registers. Table 6-5 shows the information that is provided depending on the value initially entered in the EAX register. See Section 10.1., "Processor Identification", for detailed information on the output of the CPUID instruction.

Table 6-5. Information Provided by the CPUID Instruction

Initial EAX Value	Information Provided about the Processor
0	Maximum CPUID input value. Vendor identification string ("GenuineIntel").
1	Version information (family ID, model ID, and stepping ID). Feature information (identifies the feature set for the processor model).
2	Cache information (about the processor's internal cache memory).

6.15.4. No-Operation and Undefined Instructions

The NOP (no operation) instruction increments the EIP register to point at the next instruction, but affects nothing else.

The UD2 (undefined) instruction generates an invalid opcode exception. Intel reserves the opcode for this instruction for this function. The instruction is provided to allow software to test an invalid opcode exception handler.

intel®

7

Floating-Point Unit

CHAPTER 7 FLOATING-POINT UNIT

The Intel Architecture Floating-Point Unit (FPU) provides high-performance floating-point processing capabilities. It supports the real, integer, and BCD-integer data types and the floating-point processing algorithms and exception handling architecture defined in the IEEE 754 and 854 Standards for Floating-Point Arithmetic. The FPU executes instructions from the processor's normal instruction stream and greatly improves the efficiency of Intel Architecture processors in handling the types of high-precision floating-point processing operations commonly found in scientific, engineering, and business applications.

This chapter describes the data types that the FPU operates on, the FPU's execution environment, and the FPU-specific instruction set. Detailed descriptions of the FPU instructions are given in Chapter 3, *Instruction Set Reference*, in the *Intel Architecture Software Developer's Manual, Volume 2*.

7.1. COMPATIBILITY AND EASE OF USE OF THE INTEL ARCHITECTURE FPU

The architecture of the Intel Architecture FPU has evolved in parallel with the architecture of early Intel Architecture processors. The first Intel Math Coprocessors (the Intel 8087, Intel 287, and Intel 387) were companion processors to the Intel 8086/8088, Intel 286, and Intel386 processors, respectively, and were designed to improve and extend the numeric processing capability of the Intel Architecture. The Intel486 DX processor for the first time integrated the CPU and the FPU architectures on one chip. The Pentium processor's FPU offered the same architecture as the Intel486 DX processor's FPU, but with improved performance. The Pentium Pro processor's FPU further extended the floating-point processing capability of Intel Architecture family of processors and added several new instructions to improve processing throughput.

Throughout this evolution, compatibility among the various generations of FPUs and math coprocessors has been maintained. For example, the Pentium Pro processor's FPU is fully compatible with the Pentium and Intel486 DX processors's FPUs.

Each generation of the Intel Architecture FPUs have been explicitly designed to deliver stable, accurate results when programmed using straightforward "pencil and paper" algorithms, bringing the functionality and power of accurate numeric computation into the hands of the general user. The IEEE 754 standard specifically addresses this issue, recognizing the fundamental importance of making numeric computations both easy and safe to use.

For example, some processors can overflow when two single-precision floating-point numbers are multiplied together and then divided by a third, even if the final result is a perfectly valid 32-bit number. The Intel Architecture FPUs deliver the correctly rounded result. Other typical examples of undesirable machine behavior in straightforward calculations occur when computing financial rate of return, which involves the expression $(1 + i)^n$ or when solving for roots of a quadratic equation:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If *a* does not equal 0, the formula is numerically unstable when the roots are nearly coincident or when their magnitudes are wildly different. The formula is also vulnerable to spurious over/underflows when the coefficients *a*, *b*, and *c* are all very big or all very tiny. When single-precision (4-byte) floating-point coefficients are given as data and the formula is evaluated in the FPU's normal way, keeping all intermediate results in its stack, the FPU produces impeccable single-precision roots. This happens because, by default and with no effort on the programmer's part, the FPU evaluates all those sub-expressions with so much extra precision and range as to overwhelm almost any threat to numerical integrity.

If double-precision data and results were at issue, a better formula would have to be used, and once again the FPU's default evaluation of that formula would provide substantially enhanced numerical integrity over mere double-precision evaluation.

On most machines, straightforward algorithms will not deliver consistently correct results (and will not indicate when they are incorrect). To obtain correct results on traditional machines under all conditions usually requires sophisticated numerical techniques that go beyond typical programming practice. General application programmers using straightforward algorithms will produce much more reliable programs using the Intel architectures. This simple fact greatly reduces the software investment required to develop safe, accurate computation-based products.

Beyond traditional numeric support for scientific applications, the Intel architectures have built-in facilities for commercial computing. They can process decimal numbers of up to 18 digits without round-off errors, performing **exact arithmetic** on integers as large as 2^{64} (or 10^{18}). Exact arithmetic is vital in accounting applications where rounding errors may introduce monetary losses that cannot be reconciled.

The Intel FPU's contain a number of optional numerical facilities that can be invoked by sophisticated users. These advanced features include directed rounding, gradual underflow, and programmed exception-handling facilities.

These automatic exception-handling facilities permit a high degree of flexibility in numeric processing software, without burdening the programmer. While performing numeric calculations, the processor automatically detects exception conditions that can potentially damage a calculation (for example, $X \div 0$ or $\sqrt{X}$ when $X < 0$). By default, on-chip exception logic handles these exceptions so that a reasonable result is produced and execution may proceed without program interruption. Alternatively, the processor can invoke a software exception handler to provide special results whenever various types of exceptions are detected.

7.2. REAL NUMBERS AND FLOATING-POINT FORMATS

This section describes how real numbers are represented in floating-point format in the Intel Architecture FPU. It also introduces terms such as normalized numbers, denormalized numbers, biased exponents, signed zeros, and NaNs. Readers who are already familiar with floating-point processing techniques and the IEEE standards may wish to skip this section.

7.2.1. Real Number System

As shown in Figure 7-1, the real-number system comprises the continuum of real numbers from minus infinity (–∞) to plus infinity (+∞).

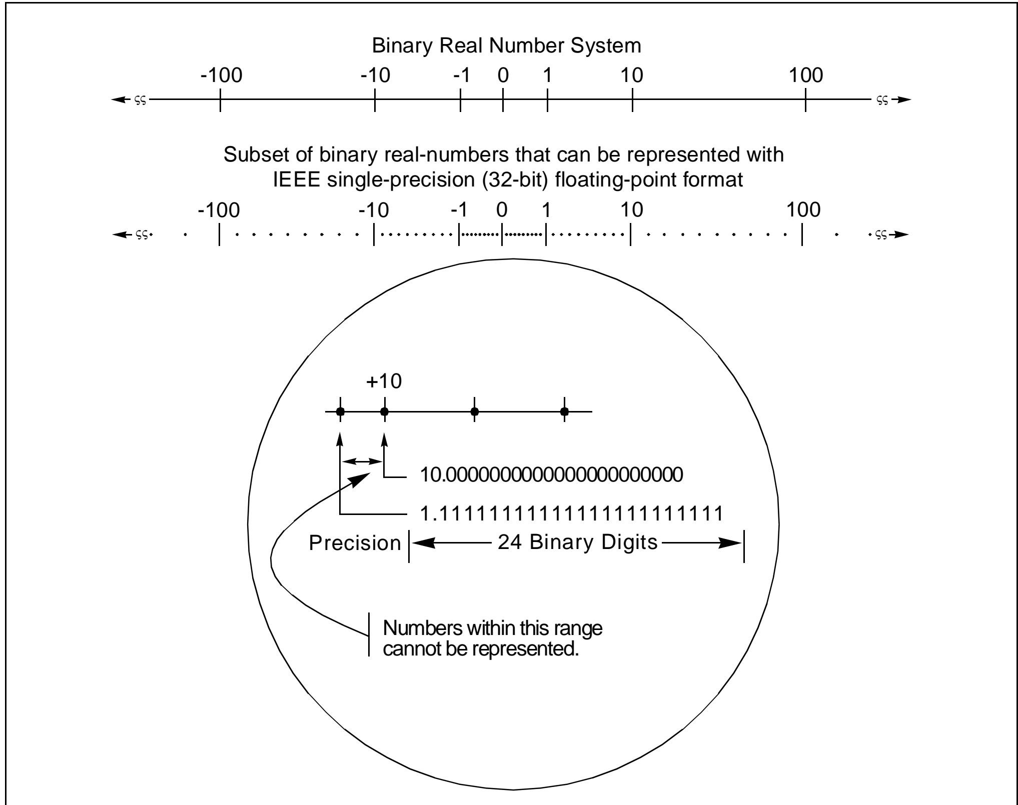


Figure 7-1. Binary Real Number System

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number calculations. As shown at the bottom of Figure 7-1, the subset of real numbers that a particular FPU supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the format that the FPU uses to represent real numbers.

7.2.2. Floating-Point Format

To increase the speed and efficiency of real-number computations, computers or FPUs typically represent real numbers in a binary floating-point format. In this format, a real number has three

parts: a sign, a significand, and an exponent. Figure 7-2 shows the binary floating-point format that the Intel Architecture FPU uses. This format conforms to the IEEE standard.

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a 1-bit binary integer (also referred to as the J-bit) and a binary fraction. The J-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power that the significand is raised to.

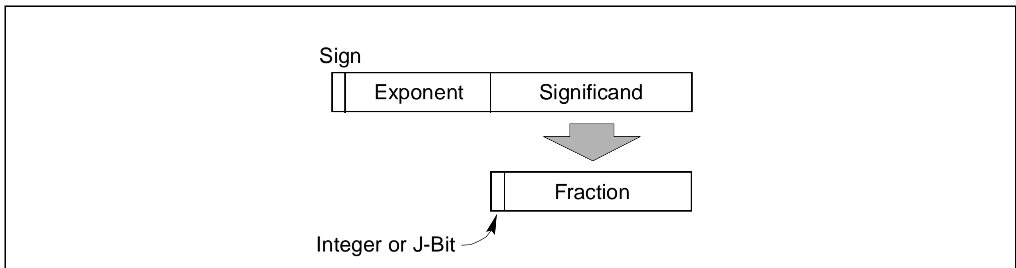


Figure 7-2. Binary Floating-Point Format

Table 7-1 shows how the real number 178.125 (in ordinary decimal format) is stored in floating-point format. The table lists a progression of real number notations that leads to the single-real, 32-bit floating-point format (which is one of the floating-point formats that the FPU supports). In this format, the significand is normalized (see Section 7.2.2.1., "Normalized Numbers") and the exponent is biased (see Section 7.2.2.2., "Biased Exponent"). For the single-real format, the biasing constant is +127.

7.2.2.1. NORMALIZED NUMBERS

In most cases, the FPU represents real numbers in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and the following fraction:

1.fff...ff

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Table 7-1. Real Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10}2$		
Scientific Binary	$1.0110010001E_{2}111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_{2}10000110$		
Single-Real Format	Sign	Biased Exponent	Normalized Significand
	0	10000110	01100100010000000000000 1. (Implied)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that specifies the number's binary point.

7.2.2.2. BIASED EXPONENT

The FPU represents exponents in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

(See Section 7.4.1., "Real Numbers" for a list of the biasing constants that the FPU uses for the various sizes of real data-types.)

7.2.3. Real Number and Non-number Encodings

A variety of real numbers and special values can be encoded in the FPU's floating-point format. These numbers and values are generally divided into the following classes:

- Signed zeros.
- Denormalized finite numbers.
- Normalized finite numbers.
- Signed infinities.
- NaNs.
- Indefinite numbers.

(The term NaN stands for "Not a Number.")

Figure 7-3 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single-precision (32-bit) format, where the term “S” indicates the sign bit, “E” the biased exponent, and “F” the fraction. (The exponent values are given in decimal.)

The FPU can operate on and/or return any of these values, depending on the type of computation being performed. The following sections describe these number and non-number classes.

7.2.3.1. SIGNED ZEROS

Zero can be represented as a +0 or a –0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been reciprocated.

7.2.3.2. NORMALIZED AND DENORMALIZED FINITE NUMBERS

Non-zero, finite numbers are divided into two classes: normalized and denormalized. The normalized finite numbers comprise all the non-zero finite values that can be encoded in a normalized real number format between zero and ∞. In the single-real format shown in Figure 7-3, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254_{10} (unbiased, the exponent range is from -126_{10} to $+127_{10}$).

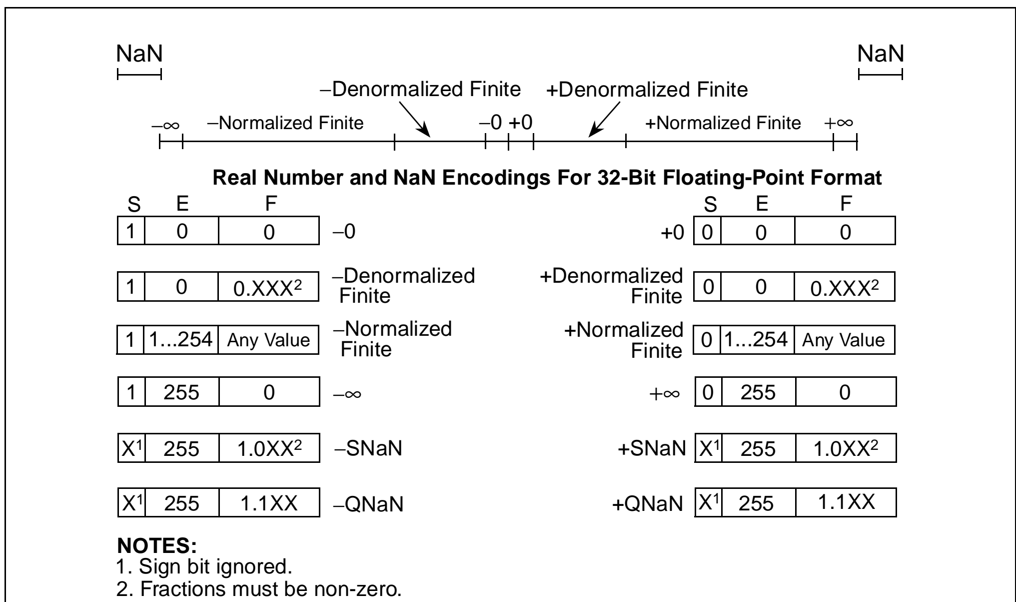


Figure 7-3. Real Numbers and NaNs

When real numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called **denormalized** (or **tiny**) numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization causes a loss of precision (the number of significant bits in the fraction is reduced by the leading zeros).

When performing normalized floating-point computations, an FPU normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an **underflow** condition.

A denormalized number is computed through a technique called gradual underflow. Table 7-2 gives an example of gradual underflow in the denormalization process. Here the single-real format is being used, so the minimum exponent (unbiased) is -126_{10}. The true result in this example requires an exponent of -129_{10} in order to have a normalized number. Since -129_{10} is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of -126_{10} is reached.

Table 7-2. Denormalization Process

Operation	Sign	Exponent*	Significand
True Result	0	–129	1.01011100000...00
Denormalize	0	–128	0.10101110000...00
Denormalize	0	–127	0.01010111000...00
Denormalize	0	–126	0.00101011100...00
Denormal Result	0	–126	0.00101011100...00

NOTE:

* Expressed as an unbiased, decimal number.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

The FPU deals with denormal values in the following ways:

- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.

When a denormal number in single- or double-real format is used as a source operand and the denormal exception is masked, the FPU automatically **normalizes** the number when it is converted to extended-real format.

7.2.3.3. SIGNED INFINITIES

The two infinities, +∞ and –∞, represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a zero significand (fraction and integer bit) and the maximum biased exponent allowed in the specified format (for example, 255_{10} for the single-real format).

The signs of infinities are observed, and comparisons are possible. Infinities are always interpreted in the affine sense; that is, –∞ is less than any finite number and +∞ is greater than any finite number. Arithmetic on infinities is always exact. Exceptions are generated only when the use of an infinity as a source operand constitutes an invalid operation.

Whereas denormalized numbers represent an underflow condition, the two infinity numbers represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

7.2.3.4. NANS

Since NaNs are non-numbers, they are not part of the real number line. In Figure 7-3, the encoding space for NaNs in the FPU floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction. (The sign bit is ignored for NaNs.)

The IEEE standard defines two classes of NaN: quiet NaNs (QNaNs) and signaling NaNs (SNaNs). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs generally signal an invalid-operation exception whenever they appear as operands in arithmetic operations. Exceptions are discussed in Section 7.7., “Floating-Point Exception Handling”.

See Section 7.6., “Operating on NaNs”, for detailed information on how the FPU handles NaNs.

7.2.4. Indefinite

For each FPU data type, one unique encoding is reserved for representing the special value **indefinite**. For example, when operating on real values, the real indefinite value is a QNaN (see Section 7.4.1., “Real Numbers”). The FPU produces indefinite values as responses to masked floating-point exceptions.

7.3. FPU ARCHITECTURE

From an abstract, architectural view, the FPU is a coprocessor that operates in parallel with the processor’s integer unit (see Figure 7-4). The FPU gets its instructions from the same instruction decoder and sequencer as the integer unit and shares the system bus with the integer unit. Other than these connections, the integer unit and FPU operate independently and in parallel. (The actual microarchitecture of an Intel Architecture processor varies among the various families of processors. For example, the Pentium Pro processor has two integer units and two FPUs; whereas, the Pentium processor has two integer units and one FPU, and the Intel486 processor has one integer unit and one FPU.)

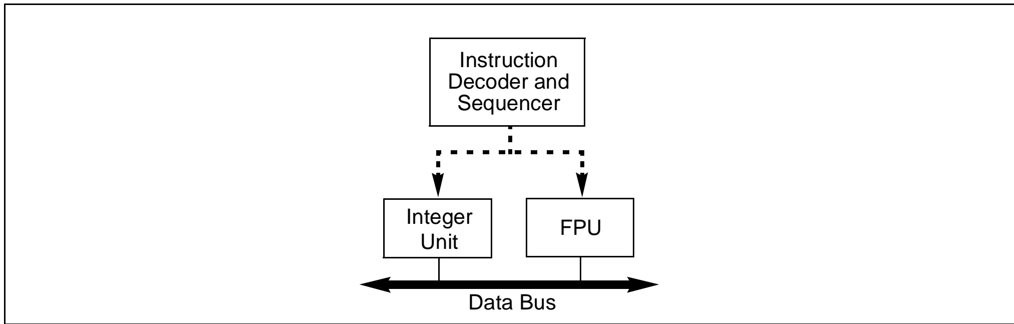


Figure 7-4. Relationship Between the Integer Unit and the FPU

The instruction execution environment of the FPU (see Figure 7-5) consists of 8 data registers (called the FPU data registers) and the following special-purpose registers:

- The status register.
- The control register.
- The tag word register.
- Instruction pointer register.
- Last operand (data pointer) register.
- Opcode register.

These registers are described in the following sections.

7.3.1. The FPU Data Registers

The FPU data registers (shown in Figure 7-5) consist of eight 80-bit registers. Values are stored in these registers in the extended-real format shown in Figure 7-17. When real, integer, or packed BCD integer values (in any of the formats shown in Figure 7-17) are loaded from memory into any of the FPU data registers, the values are automatically converted into extended-real format (if they are not already in that format). When computation results are subsequently transferred back into memory from any of the FPU registers, the results can be left in the extended-real format or converted back into one of the other FPU formats (real, integer, or packed BCD integers) shown in Figure 7-17.

The FPU instructions treat the eight FPU data registers as a register stack (see Figure 7-6). All addressing of the data registers is relative to the register on the top of the stack. The register number of the current top-of-stack register is stored in the TOP (stack TOP) field in the FPU status word. Load operations decrement TOP by one and load a value into the new top-of-stack register, and store operations store the value from the current TOP register in memory and then increment TOP by one. (For the FPU, a load operation is equivalent to a push and a store operation is equivalent to a pop.)

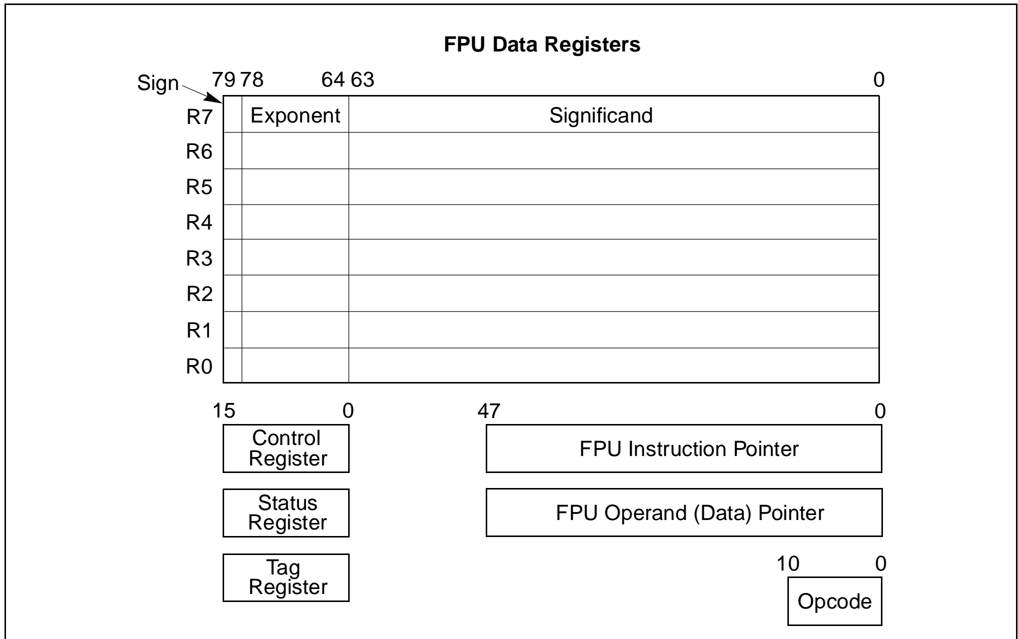


Figure 7-5. FPU Execution Environment

If a load operation is performed when TOP is at 0, register wraparound occurs and the new value of TOP is set to 7. The floating-point stack-overflow exception indicates when wraparound might cause an unsaved value to be overwritten (see Section 7.8.1.1., "Stack Overflow or Underflow Exception (#IS)").

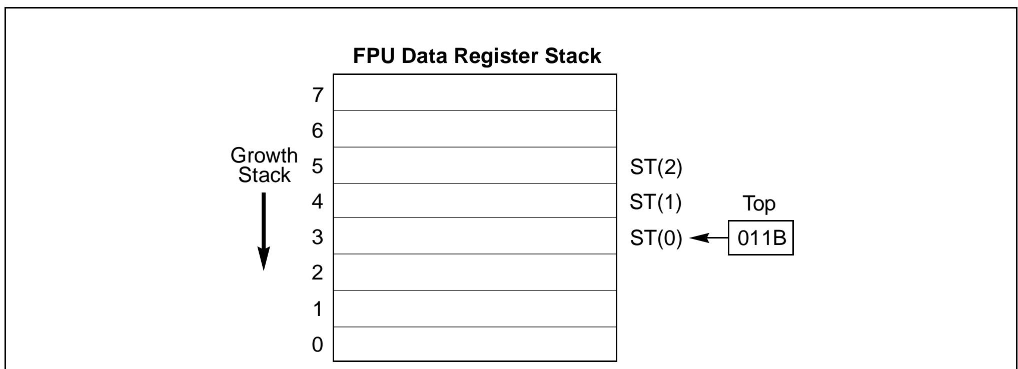


Figure 7-6. FPU Data Register Stack

Many floating-point instructions have several addressing modes that permit the programmer to implicitly operate on the top of the stack, or to explicitly operate on specific registers relative to the TOP. Assemblers supports these register addressing modes, using the expression ST(0), or simply ST, to represent the current stack top and ST(i) to specify the *i*th register from TOP in the stack ($0 \leq i \leq 7$). For example, if TOP contains 011B (register 3 is the top of the stack), the following instruction would add the contents of two registers in the stack (registers 3 and 5):

```
FADD ST, ST(2);
```

Figure 7-7 shows an example of how the stack structure of the FPU registers and instructions are typically used to perform a series of computations. Here, a two-dimensional dot product is computed, as follows:

1. The first instruction (`FLD value1`) decrements the stack register pointer (TOP) and loads the value 5.6 from memory into ST(0). The result of this operation is shown in snap-shot (a).
2. The second instruction multiplies the value in ST(0) by the value 2.4 from memory and stores the result in ST(0), shown in snap-shot (b).
3. The third instruction decrements TOP and loads the value 3.8 in ST(0).
4. The fourth instruction multiplies the value in ST(0) by the value 10.3 from memory and stores the result in ST(0), shown in snap-shot (c).
5. The fifth instruction adds the value and the value in ST(1) and stores the result in ST(0), shown in snap-shot (d).

The style of programming demonstrated in this example is supported by the floating-point instruction set. In cases where the stack structure causes computation bottlenecks, the FXCH (exchange FPU register contents) instruction can be used to streamline a computation.

7.3.1.1. PARAMETER PASSING WITH THE FPU REGISTER STACK

Like the general-purpose registers in the processor's integer unit, the contents of the FPU data registers are unaffected by procedure calls, or in other words, the values are maintained across procedure boundaries. A calling procedure can thus use the FPU data registers (as well as the procedure stack) for passing parameter between procedures. The called procedure can reference parameters passed through the register stack using the current stack register pointer (TOP) and the ST(0) and ST(i) nomenclature. It is also common practice for a called procedure to leave a return value or result in register ST(0) when returning execution to the calling procedure or program.

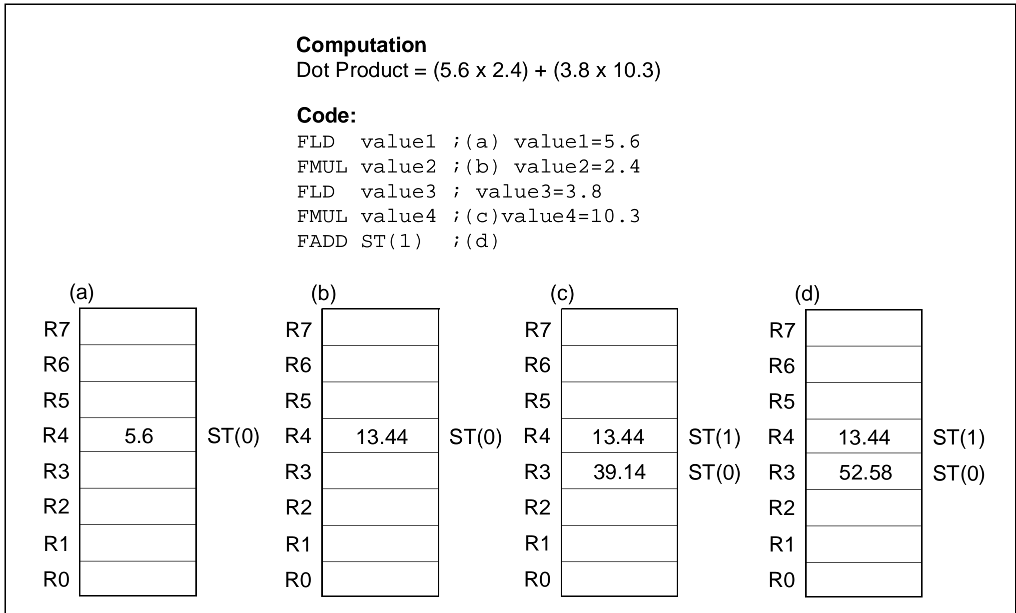


Figure 7-7. Example FPU Dot Product Computation

7.3.2. FPU Status Register

The 16-bit FPU status register (see in Figure 7-8) indicates the current state of the FPU. The flags in the FPU status register include the FPU busy flag, top-of-stack (TOP) pointer, condition code flags, error summary status flag, stack fault flag, and exception flags. The FPU sets the flags in this register to show the results of operations.

The contents of the FPU status register (referred to as the FPU status word) can be stored in memory using the FSTSW/FNSTSW, FSTENV/FNSTENV, and FSAVE/FNSAVE instructions. It can also be stored in the AX register of the integer unit, using the FSTSW/FNSTSW instructions.

7.3.2.1. TOP OF STACK (TOP) POINTER

A pointer to the FPU data register that is currently at the top of the FPU register stack is contained in bits 11 through 13 of the FPU status word. This pointer, which is commonly referred to as TOP (for top-of-stack), is a binary value from 0 to 7. See Section 7.3.1., “The FPU Data Registers”, for more information about the TOP pointer.

7.3.2.2. CONDITION CODE FLAGS

The four FPU condition code flags (C0 through C3) indicate the results of floating-point comparison and arithmetic operations. Table 7-3 summarizes the manner in which the floating-

point instructions set the condition code flags. These condition code bits are used principally for conditional branching and for storage of information used in exception handling (see Section 7.3.3., “Branching and Conditional Moves on FPU Condition Codes”).

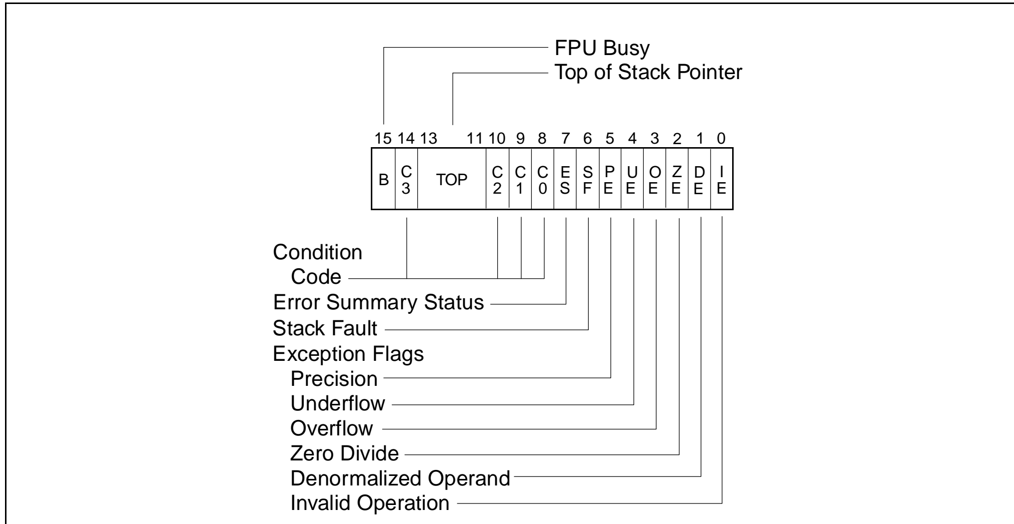


Figure 7-8. FPU Status Word

As shown in Table 7-3, the C1 condition code flag is used for a variety of functions. When both the IE and SF flags in the FPU status word are set, indicating a stack overflow or underflow exception (#IS), the C1 flag distinguishes between overflow (C1=1) and underflow (C1=0). When the PE flag in the status word is set, indicating an inexact (rounded) result, the C1 flag is set to 1 if the last rounding by the instruction was upward. The FXAM instruction sets C1 to the sign of the value being examined.

The C2 condition code flag is used by the FPREM and FPREM1 instructions to indicate an incomplete reduction (or partial remainder). When a successful reduction has been completed, the C0, C3, and C1 condition code flags are set to the three least-significant bits of the quotient (Q2, Q1, and Q0, respectively). See “FPREM1—Partial Remainder” in Chapter 3, *Instruction Set Reference*, of the *Intel Architecture Software Developer’s Manual, Volume 2*, for more information on how these instructions use the condition code flags.

The FPTAN, FSIN, FCOS, and FSINCOS instructions set the C2 flag to 1 to indicate that the source operand is beyond the allowable range of $\pm 2^{63}$.

Where the state of the condition code flags are listed as undefined in Table 7-3, do not rely on any specific value in these flags.

Table 7-3. FPU Condition Code Interpretation

Instruction	C0	C3	C2	C1
FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FTST, FUCOM, FUCOMP, FUCOMPP	Result of Comparison		Operands are not Comparable	0 or #IS
FCOMI, FCOMIP, FUCOMI, FUCOMIP	Undefined. (These instructions set the status flags in the EFLAGS register.)			#IS
FXAM	Operand class			Sign
FPREM, FPREM1	Q2	Q1	0=reduction complete 1=reduction incomplete	Q0 or #IS
F2XM1, FADD, FADDP, FBSTP, FCMOVcc, FIADD, FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, FIDIVR, FIMUL, FIST, FISTP, FISUB, FISUBR,FMUL, FMULP, FPATAN, FRNDINT, FSCALE, FST, FSTP, FSUB, FSUBP, FSUBR, FSUBRP,FSQRT, FYL2X, FYL2XP1	Undefined			Roundup or #IS
FCOS, FSIN, FSINCOS, FPTAN	Undefined		1=source operand out of range.	Roundup or #IS (Undefined if C2=1)
FABS, FBLD, FCHS, FDECSTP, FILD, FINCSTP, FLD, Load Constants, FSTP (ext. real), FXCH, FXTRACT	Undefined			0 or #IS
FLDENV, FRSTOR	Each bit loaded from memory			
FFREE, FLDCW, FCLEX/FNCLEX, FNOP, FSTCW/FNSTCW, FSTENV/FNSTENV, FSTSW/FNSTSW,	Undefined			
FINIT/FNINIT, FSAVE/FNSAVE	0	0	0	0

7.3.2.3. EXCEPTION FLAGS

The six exception flags (bits 0 through 5) of the status word indicate that one or more floating-point exceptions has been detected since the bits were last cleared. The individual exception flags (IE, DE, ZE, OE, UE, and PE) are described in detail in Section 7.7., "Floating-Point Exception Handling". Each of the exception flags can be masked by an exception mask bit in the FPU control word (see Section 7.3.4., "FPU Control Word"). The exception summary status (ES) flag (bit 7) is set when any of the unmasked exception flags are set. When the ES flag is

set, the FPU exception handler is invoked, using one of the techniques described in Section 7.7.3., "Software Exception Handling". (Note that if an exception flag is masked, the FPU will still set the flag if its associated exception occurs, but it will not set the ES flag.)

The exception flags are "sticky" bits, meaning that once set, they remain set until explicitly cleared. They can be cleared by executing the FCLEX/FNCLEX (clear exceptions) instructions, by reinitializing the FPU with the FINIT/FNINIT or FSAVE/FNSAVE instructions, or by overwriting the flags with an FRSTOR or FLDENV instruction.

The B-bit (bit 15) is included for 8087 compatibility only. It reflects the contents of the ES flag.

7.3.2.4. STACK FAULT FLAG

The stack fault flag (bit 6 of the FPU status word) indicates that stack overflow or stack underflow has occurred. The FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition. When this flag is set, the condition code flag C1 indicates the nature of the fault: overflow (C1 = 1) and underflow (C1 = 0). The SF flag is a "sticky" flag, meaning that after it is set, the processor does not clear it until it is explicitly instructed to do so (for example, by an FINIT/FNINIT, FCLEX/FNCLEX, or FSAVE/FNSAVE instruction).

See Section 7.3.6., "FPU Tag Word" for more information on FPU stack faults.

7.3.3. Branching and Conditional Moves on FPU Condition Codes

The Intel Architecture FPU (beginning with the Pentium Pro processor) supports two mechanisms for branching and performing conditional moves according to comparisons of two floating-point values. These mechanism are referred to here as the "old mechanism" and the "new mechanism."

The old mechanism is available in FPU's prior to the Pentium Pro processor and in the Pentium Pro processor. This mechanism uses the floating-point compare instructions (FCOM, FCOMP, FCOMPP, FTST, FUCOMPP, FICOM, and FICOMP) to compare two floating-point values and set the condition code flags (C0 through C3) according to the results. The contents of the condition code flags are then copied into the status flags of the EFLAGS register using a two step process (see Figure 7-9):

1. The FSTSW AX instruction moves the FPU status word into the AX register.
2. The SAHF instruction copies the upper 8 bits of the AX register, which includes the condition code flags, into the lower 8 bits of the EFLAGS register.

When the condition code flags have been loaded into the EFLAGS register, conditional jumps or conditional moves can be performed based on the new settings of the status flags in the EFLAGS register.

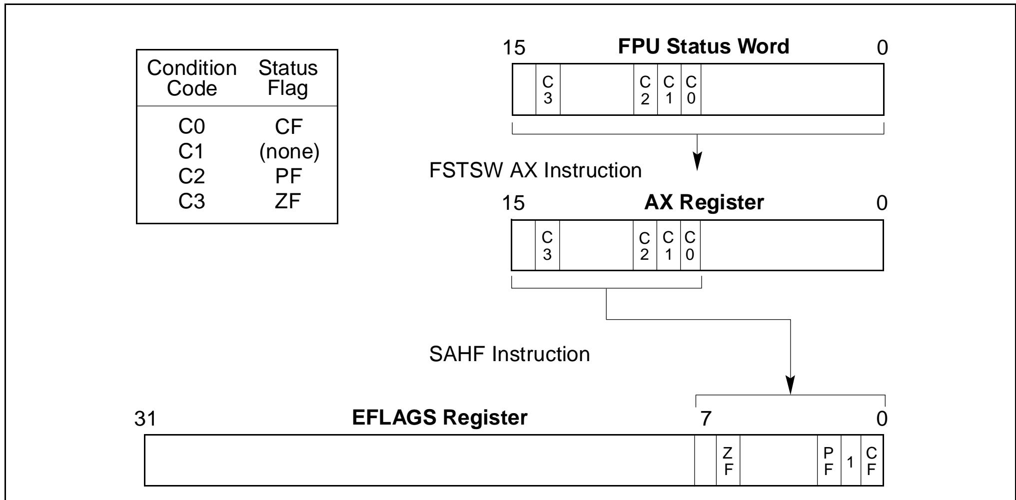


Figure 7-9. Moving the FPU Condition Codes to the EFLAGS Register

The new mechanism is available only in the Pentium Pro processor. Using this mechanism, the new floating-point compare and set EFLAGS instructions (FCOMI, FCOMIP, FUCOMI, and FUCOMIP) compare two floating-point values and set the ZF, PF, and CF flags in the EFLAGS register directly. A single instruction thus replaces the three instructions required by the old mechanism.

Note also that the FCMOV*cc* instructions (also new in the Pentium Pro processor) allow conditional moves of floating-point values (values in the FPU data registers) based on the setting of the status flags (ZF, PF, and CF) in the EFLAGS register. These instructions eliminate the need for an IF statement to perform conditional moves of floating-point values.

7.3.4. FPU Control Word

The 16-bit FPU control word (see in Figure 7-10) controls the precision of the FPU and rounding method used. It also contains the exception-flag mask bits. The control word is cached in the FPU control register. The contents of this register can be loaded with the FLDCW instruction and stored in memory with the FSTCW/FNSTCW instructions.

When the FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the FPU control word is set to 037FH, which masks all floating-point exceptions, sets rounding to nearest, and sets the FPU precision to 64 bits.

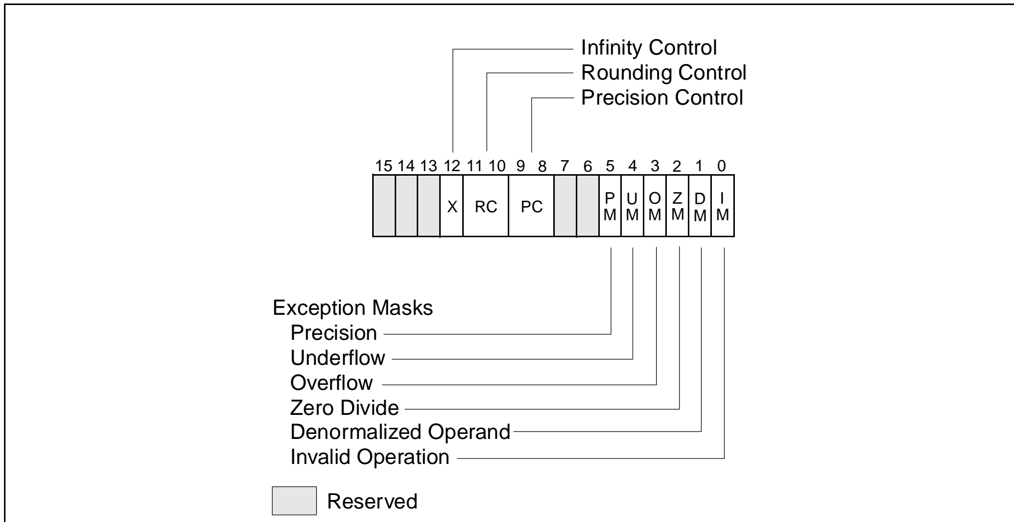


Figure 7-10. FPU Control Word

7.3.4.1. EXCEPTION-FLAG MASKS

The exception-flag mask bits (bits 0 through 5 of the FPU control word) mask the 6 exception flags in the FPU status word (also bits 0 through 5). When one of these mask bits is set, its corresponding floating-point exception is blocked from being generated.

7.3.4.2. PRECISION CONTROL FIELD

The precision-control (PC) field (bits 8 and 9 of the FPU control word) determines the precision (64, 53, or 24 bits) of floating-point calculations made by the FPU (see Table 7-4). The default precision is extended precision, which uses the full 64-bit significand available with the extended-real format of the FPU data registers. This setting is best suited for most applications, because it allows applications to take full advantage of the precision of the extended-real format.

Table 7-4. Precision Control Field (PC)

Precision	PC Field
Single Precision (24-Bits*)	00B
Reserved	01B
Double Precision (53-Bits*)	10B
Extended Precision (64-Bits)	11B

NOTE:
* Includes the implied integer bit.

The double precision and single precision settings, reduce the size of the significand to 53 bits and 24 bits, respectively. These settings are provided to support the IEEE standard and to allow exact replication of calculations which were done using the lower precision data types. Using these settings nullifies the advantages of the extended-real format's 64-bit significand length. When reduced precision is specified, the rounding of the significand value clears the unused bits on the right to zeros.

The precision-control bits only affect the results of the following floating-point instructions: FADD, FADDP, FSUB, FSUBP, FSUBR, FSUBRP, FMUL, FMULP, FDIV, FDIVP, FDIVR, FDIVRP, and FSQRT.

7.3.4.3. ROUNDING CONTROL FIELD

The rounding-control (RC) field of the FPU control register (bits 10 and 11) controls how the results of floating-point instructions are rounded. Four rounding modes are supported (see Table 7-5): round to nearest, round up, round down, and round toward zero. Round to nearest is the default rounding mode and is suitable for most applications. It provides the most accurate and statistically unbiased estimate of the true result.

Table 7-5. Rounding Control Field (RC)

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero).
Round down (toward $-\infty$)	01B	Rounded result is close to but no greater than the infinitely precise result.
Round up (toward $+\infty$)	10B	Rounded result is close to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is close to but no greater in absolute value than the infinitely precise result.

The round up and round down modes are termed **directed rounding** and can be used to implement interval arithmetic. Interval arithmetic is used to determine upper and lower bounds for the true result of a multistep computation, when the intermediate results of the computation are subject to rounding.

The round toward zero mode (sometimes called the “chop” mode) is commonly used when performing integer arithmetic with the FPU.

Whenever possible, the FPU produces an infinitely precise result in the destination format (single, double, or extended real). However, it is often the case that the infinitely precise result of an arithmetic or store operation cannot be encoded exactly in the format of the destination operand.

For example, the following value (*a*) has a 24-bit fraction. The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the single-real format (which has only a 23-bit fraction):

(*a*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 011\underline{1}E_2\ 101$

To round this result (*a*), the FPU first selects two representable fractions *b* and *c* that most closely bracket *a* in value ($b < a < c$).

(*b*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 011E_2\ 101$

(*c*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 100E_2\ 101$

The FPU then sets the result to *b* or to *c* according to the rounding mode selected in the RC field. Rounding introduces an error in a result that is less than one unit in the last place to which the result is rounded.

The rounded result is called the inexact result. When the FPU produces an inexact result, the floating-point precision (inexact) flag (PE) is set in the FPU status word.

When the overflow exception is masked and the infinitely precise result is between the largest positive finite value allowed in a particular format and +∞, the FPU rounds the result as shown in Table 7-6.

Table 7-6. Rounding of Positive Numbers With Masked Overflow

Rounding Mode	Result
Rounding to nearest (even)	+∞
Rounding toward zero (Truncate)	Maximum, positive finite value
Rounding up (toward +∞)	+∞
Rounding down) (toward –∞)	Maximum, positive finite value

When the overflow exception is masked and the infinitely precise result is between the largest negative finite value allowed in a particular format and –∞, the FPU rounds the result as shown in Table 7-7.

Table 7-7. Rounding of Negative Numbers With Masked Overflow

Rounding Mode	Result
Rounding to nearest (even)	–∞
Rounding toward zero (Truncate)	Maximum, negative finite value
Rounding up (toward +∞)	Maximum, negative finite value
Rounding down) (toward –∞)	–∞

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

7.3.5. Infinity Control Flag

The infinity control flag (bit 12 of the FPU control word) is provided for compatibility with the Intel 287 Math Coprocessor; it is not meaningful for the Pentium Pro processor FPU or for the Pentium processor FPU, the Intel486 processor FPU, or Intel 387 processor NPX. See Section 7.2.3.3., "Signed Infinities", for information on how the Intel Architecture FPUs handle infinity values.

7.3.6. FPU Tag Word

The 16-bit tag word (see in Figure 7-11) indicates the contents of each the 8 registers in the FPU data-register stack (one 2-bit tag per register). The tag codes indicate whether a register contains a valid number, zero, or a special floating-point number (NaN, infinity, denormal, or unsupported format), or whether it is empty. The FPU tag word is cached in the FPU in the FPU tag word register. When the FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the FPU tag word is set to FFFFH, which marks all the FPU data registers as empty.

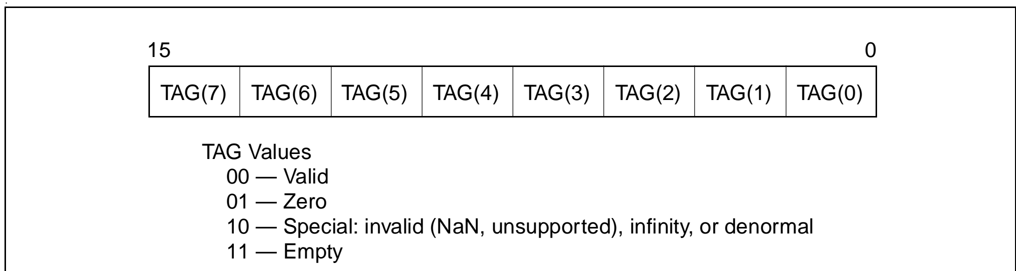


Figure 7-11. FPU Tag Word

Each tag in the FPU tag word corresponds to a physical register (numbers 0 through 7). The current top-of-stack (TOP) pointer stored in the FPU status word can be used to associate tags with registers relative to ST(0).

The FPU uses the tag values to detect stack overflow and underflow conditions. Stack overflow occurs when the TOP pointer is decremented (due to a register load or push operation) to point to a non-empty register. Stack underflow occurs when the TOP pointer is incremented (due to a save or pop operation) to point to an empty register or when an empty register is also referenced as a source operand. A non-empty register is defined as a register containing a zero (01), a valid value (00), or an special (10) value.

Application programs and exception handlers can use this tag information to check the contents of an FPU data register without performing complex decoding of the actual data in the register. To read the tag register, it must be stored in memory using either the FSTENV/FNSTENV or FSAVE/FNSAVE instructions. The location of the tag word in memory after being saved with one of these instructions is shown in Figures 7-13 through 7-16.

Software cannot directly load or modify the tags in the tag register. The FLDENV and FRSTOR instructions load an image of the tag register into the FPU; however, the FPU uses those tag

values only to determine if the data registers are empty (11B) or non-empty (00B, 01B, or 10B). If the tag register image indicates that a data register is empty, the tag in the tag register for that data register is marked empty (11B); if the tag register image indicates that the data register is non-empty, the FPU reads the actual value in the data register and sets the tag for the register accordingly. This action prevents a program from setting the values in the tag register to incorrectly represent the actual contents of non-empty data registers.

7.3.7. The FPU Instruction and Operand (Data) Pointers

The FPU stores pointers to the instruction and operand (data) for the last non-control instruction executed in two 48-bit registers: the FPU instruction pointer and FPU operand (data) pointer registers (see Figure 7-5). (This information is saved to provide state information for exception handlers.)

The contents of the FPU instruction and operand pointer registers remain unchanged when any of the control instructions (FINIT/FNINIT, FCLEX/FNCLEX, FLDCW, FSTCW/FNSTCW, FSTSW/FNSTSW, FSTENV/FNSTENV, FLDENV, FSAVE/FNSAVE, FRSTOR, and WAIT/FWAIT) are executed. The contents of the FPU operand register are undefined if the prior non-control instruction did not have a memory operand.

The pointers stored in the FPU instruction and operand pointer registers consist of an offset (stored in bits 0 through 31) and a segment selector (stored in bits 32 through 47).

These registers can be accessed by the FSTENV/FNSTENV, FLDENV, FINIT/FNINIT, FSAVE/FNSAVE and FRSTOR instructions. The FINIT/FNINIT and FSAVE/FNSAVE instructions clear these registers.

For all the Intel Architecture FPUs and NPXs except the 8087, the FPU instruction pointer points to any prefixes that preceded the instruction. For the 8087, the FPU instruction pointer points only to the actual opcode.

7.3.8. Last Instruction Opcode

The FPU stores the opcode of the last non-control instruction executed in an 11-bit FPU opcode register. (This information provides state information for exception handlers.) Only the first and second opcode bytes (after all prefixes) are stored in the FPU opcode register. Figure 7-12 shows the encoding of these two bytes. Since the upper 5 bits of the first opcode byte are the same for all floating-point opcodes (11011B), only the lower 3 bits of this byte are stored in the opcode register.

7.3.9. Saving the FPU's State

The FSTENV/FNSTENV and FSAVE/FNSAVE instructions store FPU state information in memory for use by exception handlers and other system and application software. The FSTENV/FNSTENV instruction saves the contents of the status, control, tag, FPU instruction pointer, FPU operand pointer, and opcode registers. The FSAVE/FNSAVE instruction stores that

information plus the contents of the FPU data registers. Note that the FSAVE/FNSAVE instruction also initializes the FPU to default values (just as the FINIT/FNINIT instruction does) after it has saved the original state of the FPU.

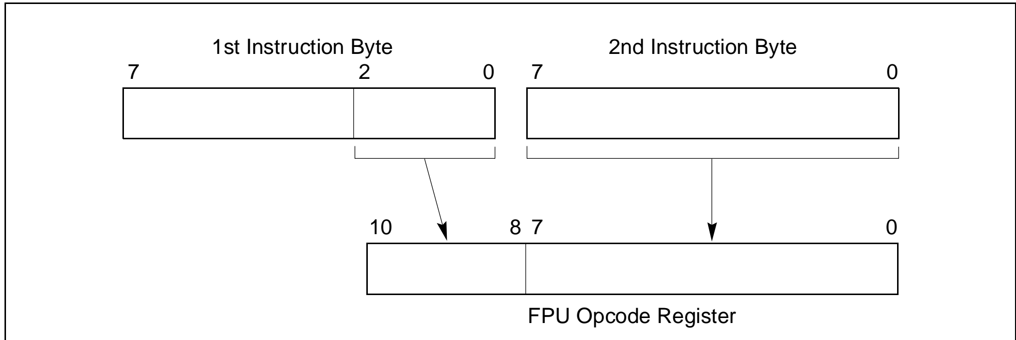


Figure 7-12. Contents of FPU Opcode Registers

The manner in which this information is stored in memory depends on the operating mode of the processor (protected mode or real-address mode) and on the operand-size attribute in effect (32-bit or 16-bit). See Figures 7-13 through 7-16. In virtual-8086 mode or SMM, the real-address mode formats shown in Figure 7-16 is used. See “Using the FPU in SMM” in Chapter 11 of the *Intel Architecture Software Developer’s Manual, Volume 3*, for special considerations for using the FPU while in SMM.

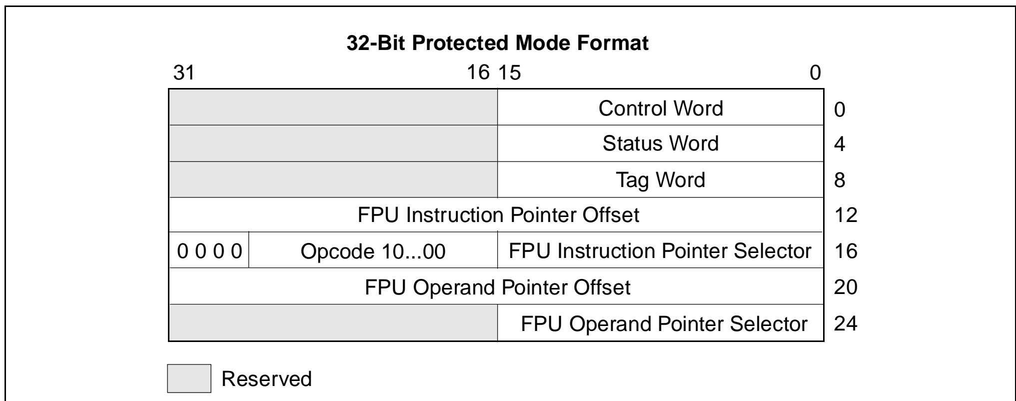


Figure 7-13. Protected Mode FPU State Image in Memory, 32-Bit Format

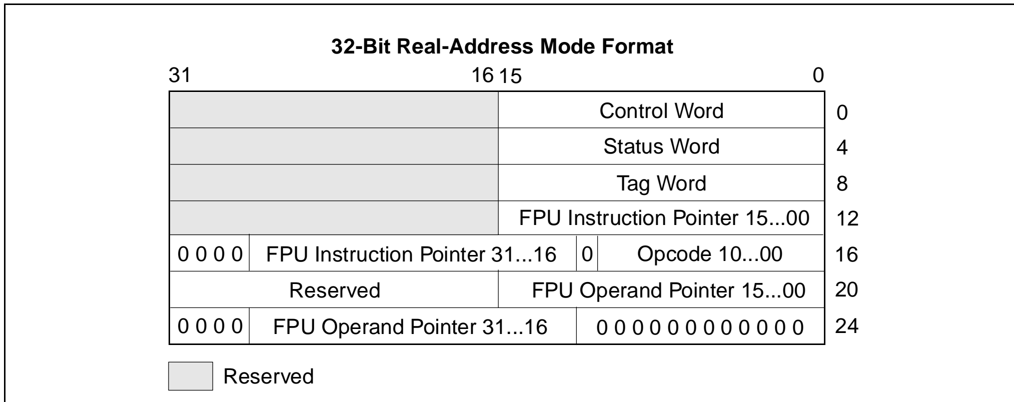


Figure 7-14. Real Mode FPU State Image in Memory, 32-Bit Format

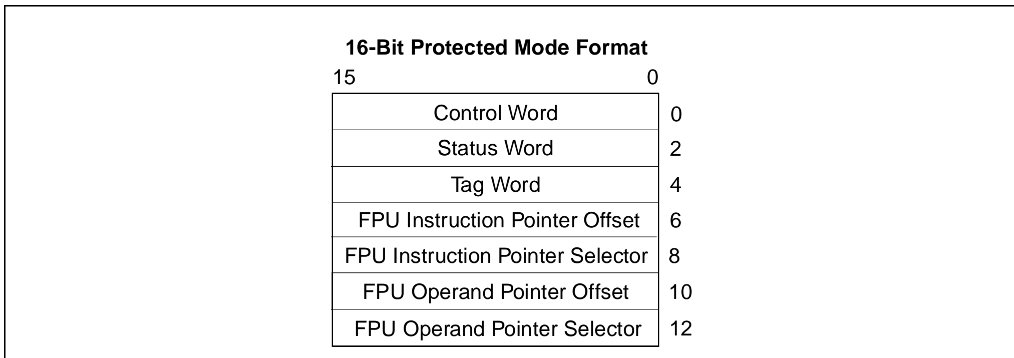


Figure 7-15. Protected Mode FPU State Image in Memory, 16-Bit Format

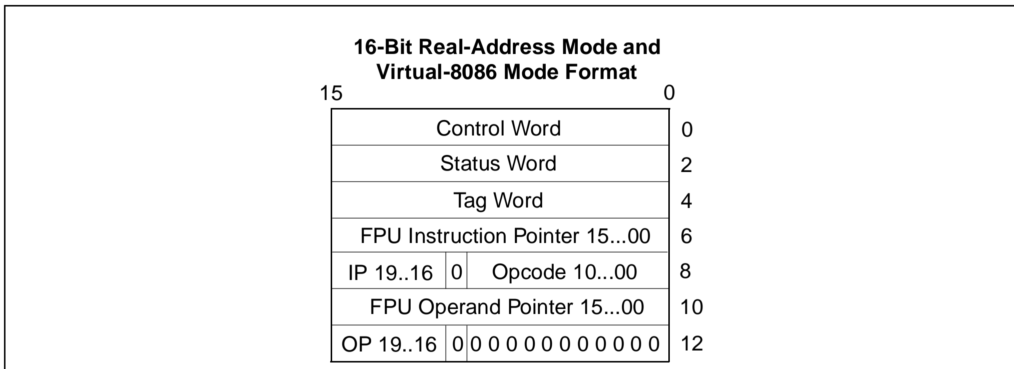


Figure 7-16. Real Mode FPU State Image in Memory, 16-Bit Format

The FLDENV and FRSTOR instructions allow FPU state information to be loaded from memory into the FPU. Here, the FLDENV instruction loads only the status, control, tag, FPU instruction pointer, FPU operand pointer, and opcode registers, and the FRSTOR instruction loads all the FPU registers, including the FPU stack registers.

7.4. FLOATING-POINT DATA TYPES AND FORMATS

The Intel Architecture FPU recognizes and operates on seven data types, divided into three groups: reals, integers, and packed BCD integers. Figure 7-17 shows the data formats for each of the FPU data types. Table 7-8 gives the length, precision, and approximate normalized range that can be represented of each FPU data type. Denormal values are also supported in each of the real types, as required by IEEE Std. 854.

With the exception of the 80-bit extended-real format, all of these data types exist in memory only. When they are loaded into FPU data registers, they are converted into extended-real format and operated on in that format.

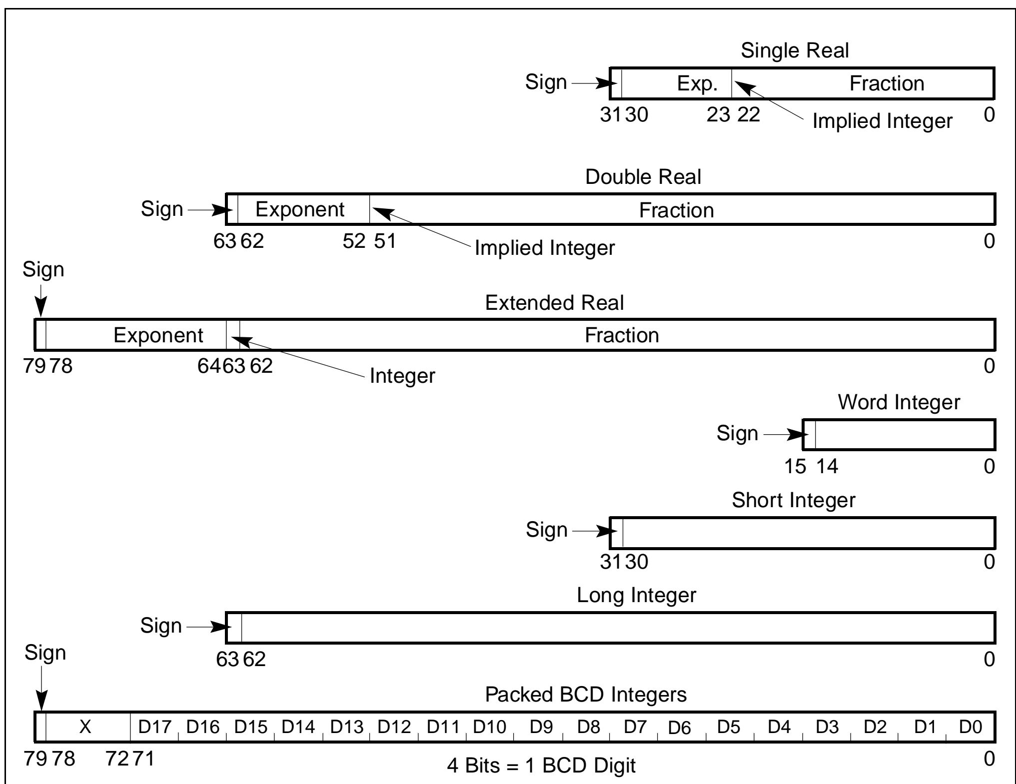


Figure 7-17. Floating-Point Unit Data Type Formats

When stored in memory, the least significant byte of an FPU data-type value is stored at the initial address specified for the value. Successive bytes from the value are then stored in successively higher addresses in memory. The floating-point instructions load and store memory operands using only the initial address of the operand.

7.4.1. Real Numbers

The FPU's three real data types (single-real, double-real, and extended-real) correspond directly to the single-precision, double-precision, and double-extended-precision formats in the IEEE standard. The extended-precision format is the format used by the data registers in the FPU. Table 7-8 gives the precision and range of these data types and Figure 7-17 gives the formats.

For the single-real and double-real formats, only the fraction part of the significand is encoded. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers. For the extended-real format, the integer is contained in bit 63, and the most-significant fraction bit is bit 62. Here, the integer is explicitly set to 1 for normalized numbers, infinities, and NaNs, and to 0 for zero and denormalized numbers.

Table 7-8. Length, Precision, and Range of FPU Data Types

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Binary Real				
Single real	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}
Double real	64	53	2^{-1022} to 2^{1023}	2.23×10^{-308} to 1.79×10^{308}
Extended real	80	64	2^{-16382} to 2^{16383}	3.37×10^{-4932} to 1.18×10^{4932}
Binary Integer				
Word integer	16	15	-2^{15} to $2^{15} - 1$	–32,768 to 32,767
Short integer	32	31	-2^{31} to $2^{31} - 1$	-2.14×10^{9} to 2.14×10^{9}
Long integer	64	63	-2^{63} to $2^{63} - 1$	-9.22×10^{18} to 9.22×10^{18}
Packed BCD Integers	80	18 (decimal digits)	Not Pertinent	$(-10^{18} + 1)$ to $(10^{18} - 1)$

The exponent of each real data type is encoded in biased format. The biasing constant is 127 for the single-real format, 1023 for the double-real format, and 16,383 for the extended-real format.

Table 7-9 shows the encodings for all the classes of real numbers (that is, zero, denormalized-finite, normalized-finite, and ∞) and NaNs for each of the three real data-types. It also gives the format for the real indefinite value.

When storing real values in memory, single-real values are stored in 4 consecutive bytes in memory; double-real values are stored in 8 consecutive bytes; and extended-real values are stored in 10 consecutive bytes.

As a general rule, values should be stored in memory in double-real format. This format provides sufficient range and precision to return correct results with a minimum of programmer attention.

The single-real format is appropriate for applications that are constrained by memory; however, it provides less precision and a greater chance of overflow. The single-real format is also useful for debugging algorithms, because rounding problems will manifest themselves more quickly in this format. The extended-real format is normally reserved for holding intermediate results in the FPU registers and constants. Its extra length is designed to shield final results from the effects of rounding and overflow/underflow in intermediate calculations. However, when an application requires the maximum range and precision of the FPU (for data storage, computations, and results), values can be stored in memory in extended-real format.

The real indefinite value is a QNaN encoding that is stored by several floating-point instructions in response to a masked floating-point invalid-operation exception (see Table 7-20).

Table 7-9. Real Number and NaN Encodings

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
Positive	+∞	0	11..11	1	00..00
	+Normals	0 . . 0	11..10 . . 00..01	1 . . 1	11..11 . . 00..00
	+Denormals	0 . . 0	00..00 . . 00..00	0 . . 0	11.11 . . 00..01
	+Zero	0	00..00	0	00..00
Negative	–Zero	1	00..00	0	00..00
	–Denormals	1 . . 1	00..00 . . 00..00	0 . . 0	00..01 . . 11..11
	–Normals	1 . . 1	00..01 . . 11..10	1 . . 1	00..00 . . 11..11
	–∞	1	11..11	1	00..00
NaNs	SNaN	X	11..11	1	0X..XX[2]
	QNaN	X	11..11	1	1X..XX
	Real Indefinite (QNaN)	1	11..11	1	10..00
	Single-Real: Double-Real: Extended-Real		← 8 Bits → ← 11 Bits → ← 15 Bits →		← 23 Bits → ← 52 Bits → ← 63 Bits →

NOTES:

1. Integer bit is implied and not stored for single-real and double-real formats.
2. The fraction for SNaN encodings must be non-zero.

7.4.2. Binary Integers

The FPU's three binary integer data types (word, short, and long) have identical formats, except for length. Table 7-8 gives the precision and range of these data types and Figure 7-17 gives the formats. Table 7-10 gives the encodings of the three binary integer types.

Table 7-10. Binary Integer Encodings

Class		Sign	Magnitude
Positive	Largest	0	11..11
		.	.
		.	.
		.	.
	Smallest	0	00..01
Zero		0	00..00
Negative	Smallest	1	11..11
		.	.
		.	.
		.	.
		.	.
	Largest	1	00..00
Integer Indefinite		1	00..00
		Word Integer:	← 15 bits →
		Short Integer:	← 31 Bits →
		Long Integer:	← 63 Bits →

The most significant bit of each format is the sign bit (0 for positive and 1 for negative). Negative values are represented in standard two's complement notation. The quantity zero is represented with all bits (including the sign bit) set to zero. Note that the FPU's word-integer data type is identical to the word-integer data type used by the processor's integer unit and the short-integer format is identical to the integer unit's doubleword-integer data type.

Word-integer values are stored in 2 consecutive bytes in memory; short-integer values are stored in 4 consecutive bytes; and long-integer values are stored in 8 consecutive bytes. When loaded into the FPU's data registers, all the binary integers are exactly representable in the extended-real format.

The binary integer encoding 100..00B represents either of two things, depending on the circumstances of its use:

- The largest negative number supported by the format (-2^{15}, -2^{31}, or -2^{63}).
- The **integer indefinite** value.

If this encoding is used as a source operand (as in an integer load or integer arithmetic instruction), the FPU interprets it as the largest negative number representable in the format being used. If the FPU detects an invalid operation when storing an integer value in memory with an FIST/FISTP instruction and the invalid-operation exception is masked, the FPU stores the integer indefinite encoding in the destination operand as a masked response to the exception. In situations where the origin of a value with this encoding may be ambiguous, the invalid-operation exception flag can be examined to see if the value was produced as a response to an exception.

If the integer indefinite is stored in memory and is later loaded back into an FPU data register, it is interpreted as the largest negative number supported by the format.

7.4.3. Decimal Integers

Decimal integers are stored in a 10-byte, packed BCD format. Table 7-8 gives the precision and range of this data type and Figure 7-17 shows the format. In this format, the first 9 bytes hold 18 BCD digits, 2 digits per byte (see Section 5.2.3., "BCD Integers"). The least-significant digit is contained in the lower half-byte of byte 0 and the most-significant digit is contained in the upper half-byte of byte 9. The most significant bit of byte 10 contains the sign bit (0 = positive and 1 = negative). (Bits 0 through 6 of byte 10 are don't care bits.) Negative decimal integers are not stored in two's complement form; they are distinguished from positive decimal integers only by the sign bit.

Table 7-11 gives the possible encodings of value in the decimal integer data type.

The decimal integer format exists in memory only. When a decimal integer is loaded in a data register in the FPU, it is automatically converted to the extended-real format. All decimal integers are exactly representable in extended-real format.

The **packed decimal indefinite** encoding is stored by the FBSTP instruction in response to a masked floating-point invalid-operation exception. Attempting to load this value with the FBLD instruction produces an undefined result.

7.4.4. Unsupported Extended-Real Encodings

The extended-real format permits many encodings that do not fall into any of the categories shown in Table 7-9. Table 7-12 shows these unsupported encodings. Some of these encodings were supported by the Intel 287 math coprocessor; however, most of them are not supported by the Intel 387 math coprocessor, or the internal FPUs in the Intel486, Pentium, or Pentium Pro processors. These encodings are no longer supported due to changes made in the final version of IEEE Std. 754 that eliminated these encodings.

The categories of encodings formerly known as pseudo-NaNs, pseudo-infinities, and un-normal numbers are not supported. The Intel 387 math coprocessor and the internal FPUs in the Intel486, Pentium, and Pentium Pro processors generate the invalid-operation exception when they are encountered as operands.

The encodings formerly known as pseudo-denormal numbers are not generated by the Intel 387 math coprocessor and the internal FPUs in the Intel486, Pentium, and Pentium Pro processors; however, they are used correctly when encountered as operands. The exponent is treated as if it were 00..01B and the mantissa is unchanged. The denormal exception is generated.

Table 7-11. Packed Decimal Integer Encodings

Class	Sign		Magnitude					
			digit	digit	digit	digit	...	digit
Positive Largest	0	0000000	1001	1001	1001	1001	...	1001
	.	.			.			
	.	.			.			
Smallest	0	0000000	0000	0000	0000	0000	...	0001
Zero	0	0000000	0000	0000	0000	0000	...	0000
Negative Zero	1	0000000	0000	0000	0000	0000	...	0000
Smallest	1	0000000	0000	0000	0000	0000	...	0001
	.	.			.			
	.	.			.			
Largest	1	0000000	1001	1001	1001	1001	...	1001
Decimal Integer Indefinite	1	1111111	1111	1111	UUUU*	UUUU	...	UUUU
	← 1 byte →		← 9 bytes →					

NOTE:
* UUUU means bit values are undefined and may contain any value.

7.5. FPU INSTRUCTION SET

The floating-point instructions that the Intel Architecture FPU supports can be grouped into six functional categories:

- Data transfer instructions
- Basic arithmetic instructions
- Comparison instructions
- Transcendental instructions
- Load constant instructions
- FPU control instructions

See Section 6.2.3., “Floating-Point Instructions”, for a list of the floating-point instructions by category.

The following section briefly describes the instructions in each category. Detailed descriptions of the floating-point instructions are given in Chapter 3, *Instruction Set Reference*, in the *Intel Architecture Software Developer’s Manual, Volume 2*.

Table 7-12. Unsupported Extended-Real Encodings

Class		Sign	Biased Exponent	Significand	
				Integer	Fraction
Positive Pseudo-NaNs	Quiet	0 . 0	11..11 . 11..11	0	11..11 . 10..00
	Signaling	0 . 0	11..11 . 11..11	0	01..11 . 00..01
Positive Reals	Pseudo-infinity	0	11..11	0	00..00
	Unnormals	0 . 0	11..10 . 00..01	0	11..11 . 00..00
	Pseudo-denormals	0 . 0	00..00 . 00..00	1	11..11 . 00..00
Negative Reals	Pseudo-denormals	1 . 1	00..00 . 00..00	1	11..11 . 00..00
	Unnormals	1 . 1	11..10 . 00..01	0	11..01 . 00..00
	Pseudo-infinity	1	11..11	0	00..00
Negative Pseudo-NaNs	Signaling	1 . 1	11..11 . 11..11	0	01..11 . 00..01
	Quiet	1 . 1	11..11 . 11..11	0	11..11 . 10..00
			← 15 bits →		← 63 bits →

7.5.1. Escape (ESC) Instructions

All of the instructions in the FPU instruction set fall into a class of instructions known as escape (ESC) instructions. All of these instructions have a common opcode format, which is slightly different from the format used by the integer and operating-system instructions.

7.5.2. FPU Instruction Operands

Most floating-point instructions require one or two operands, which are located on the FPU data-register stack or in memory. (None of the floating-point instructions accept immediate operands.)

When an operand is located in a data register, it is referenced relative to the ST(0) register (the register at the top of the register stack), rather than by a physical register number. Often the ST(0) register is an implied operand.

Operands in memory can be referenced using the same operand addressing methods available for the integer and system instructions.

7.5.3. Data Transfer Instructions

The data transfer instructions (see Table 7-13) perform the following operations:

- Load real, integer, or packed BCD operands from memory into the ST(0) register.
- Store the value in the ST(0) register in memory in real, integer, or packed BCD format.
- Move values between registers in the FPU register stack.

Table 7-13. Data Transfer Instructions

Real		Integer		Packed Decimal	
FLD	Load Real	FILD	Load Integer	FBLD	Load Packed Decimal
FST	Store Real	FIST	Store Integer		
FSTP	Store Real and Pop	FISTP	Store Integer and Pop	FBSTP	Store Packed Decimal and Pop
FXCH	Exchange Register Contents				
FCMOV*cc*	Conditional Move				

Operands are normally stored in the FPU data registers in extended-real format (see Section 7.3.4.2., "Precision Control Field"). The FLD (load real) instruction pushes a real operand from memory onto the top of the FPU data-register stack. If the operand is in single- or double-real format, it is automatically converted to extended-real format. This instruction can also be used to push the value in a selected FPU data register onto the top of the register stack.

The FILD (load integer) instruction converts an integer operand in memory into extended-real format and pushes the value onto the top of the register stack. The FBLD (load packed decimal) instruction performs the same load operation for a packed BCD operand in memory.

The FST (store real) and FIST (store integer) instructions store the value in register ST(0) in memory in the destination format (real or integer, respectively). Again, the format conversion is carried out automatically.

The FSTP (store real and pop), FISTP (store integer and pop), and FBSTP (store packed decimal and pop) instructions store the value in the ST(0) registers into memory in the destination format (real, integer, or packed BCD), then performs a **pop** operation on the register stack. A pop operation causes the ST(0) register to be marked empty and the stack pointer (TOP) in the FPU control work to be incremented by 1. The FSTP instruction can also be used to copy the value in the ST(0) register to another FPU register [ST(i)].

The FXCH (exchange register contents) instruction exchanges the value in a selected register in the stack [ST(i)] with the value in ST(0).

The FCMOV*cc* (conditional move) instructions move the value in a selected register in the stack [ST(i)] to register ST(0). These instructions move the value only if the conditions specified with a condition code (*cc*) are satisfied (see Table 7-14). The conditions being tested with the FCMOV*cc* instructions are represented by the status flags in the EFLAGS register. The condition code mnemonics are appended to the letters "FCMOV" to form the mnemonic for a FCMOV*cc* instruction.

Table 7-14. Floating-Point Conditional Move Instructions

Instruction Mnemonic	Status Flag States	Condition Description
FCMOVB	CF=1	Below
FCMOVNB	CF=0	Not below
FCMOVE	ZF=1	Equal
FCMOVNE	ZF=0	Not equal
FCMOVBE	(CF or ZF)=1	Below or equal
FCMOVNBE	(CF or ZF)=0	Not below nor equal
FCMOVU	PF=1	Unordered
FCMOVNU	PF=0	Not unordered

Like the CMOV*cc* instructions, 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.

NOTE

The FCMOV*cc* instructions may not be supported on some processors in the Pentium Pro processor family. Software can check if the FCMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "CPUID—CPU Identification" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*).

7.5.4. Load Constant Instructions

The following instructions push commonly used constants onto the top [ST(0)] of the FPU register stack:

FLDZ	Load +0.0
FLD1	Load +1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

The constant values have full extended-real precision (64 bits) and are accurate to approximately 19 decimal digits. They are stored internally in a format more precise than extended real. When loading the constant, the FPU rounds the more precise internal constant according to the RC (rounding control) field of the FPU control word. See Section 7.5.8., "Pi", for information on the π constant.

7.5.5. Basic Arithmetic Instructions

The following floating-point instructions perform basic arithmetic operations on real numbers. Where applicable, these instructions match IEEE Standard 754:

FADD/FADDP	Add real
FIADD	Add integer to real
FSUB/FSUBP	Subtract real
FISUB	Subtract integer from real
FSUBR/FSUBRP	Reverse subtract real
FISUBR	Reverse subtract real from integer
FMUL/FMULP	Multiply real
FIMUL	Multiply integer by real
FDIV/FDIVP	Divide real
FIDIV	Divide real by integer
FDIVR/FDIVRP	Reverse divide
FIDIVR	Reverse divide integer by real
FABS	Absolute value
FCHS	Change sign
FSQRT	Square root
FPREM	Partial remainder
FPREM1	IEEE partial remainder
FRNDINT	Round to integral value
FXTRACT	Extract exponent and significand

The add, subtract, multiply and divide instructions operate on the following types of operands:

- Two FPU register values.
- A register value and a real or integer value in memory.

Operands in memory can be in single-real, double-real, short-integer, or word-integer format. They are converted to extended-real format automatically.

Reverse versions of the subtract and divide instructions are provided to foster efficient coding. For example, the FSUB instruction subtracts the value in a specified FPU register [ST(i)] from the value in register ST(0); whereas, the FSUBR instruction subtracts the value in ST(0) from the value in ST(i). The results of both operations are stored in register ST(0). These instructions eliminate the need to exchange values between register ST(0) and another FPU register to perform a subtraction or division.

The pop versions of the add, subtract, multiply and divide instructions pop the FPU register stack following the arithmetic operation.

The FPREM instruction computes the remainder from the division of two operands in the manner used by the Intel 8087 and Intel 287 math coprocessors; the FPREM1 instructions computes the remainder is the manner specified in the IEEE specification.

The FSQRT instruction computes the square root of the source operand.

The FRNDINT instructions rounds a real value to its nearest integer value, according to the current rounding mode specified in the RC field of the FPU control word. This instruction performs a function similar to the FIST/FISTP instructions, except that the result is saved in a real format.

The FABS, FCHS, and FXTRACT instructions perform convenient arithmetic operations. The FABS instruction produces the absolute value of the source operand. The FCHS instruction changes the sign of the source operand. The FXTRACT instruction separates the source operand into its exponent and fraction and stores each value in a register in real format.

7.5.6. Comparison and Classification Instructions

The following instructions compare or classify real values:

FCOM/FCOMP/FCOMPP	Compare real and set FPU condition code flags.
FUCOM/FUCOMP/FUCOMPP	Unordered compare real and set FPU condition code flags.
FICOM/FICOMP	Compare integer and set FPU condition code flags.
FCOMI/FCOMIP	Compare real and set EFLAGS status flags.
FUCOMI/FUCOMIP	Unordered compare real and set EFLAGS status flags.
FTST	Test (compare real with 0.0).
FXAM	Examine.

Comparison of real values differ from comparison of integers because real values have four (rather than three) mutually exclusive relationships: less than, equal, greater than, and unordered.

The unordered relationship is true when at least one of the two values being compared is a NaN or in an undefined format. This additional relationship is required because, by definition, NaNs are not numbers, so they cannot have less than, equal, or greater than relationships with other real values.

The FCOM, FCOMP, and FCOMPP instructions compare the value in register ST(0) with a real source operand and set the condition code flags (C0, C2, and C3) in the FPU status word according to the results (see Table 7-15). If an unordered condition is detected (one or both of the values is a NaN or in an undefined format), a floating-point invalid-operation exception is generated.

The pop versions of the instruction pop the FPU register stack once or twice after the comparison operation is complete.

The FUCOM, FUCOMP, and FUCOMPP instructions operate the same as the FCOM, FCOMP, and FCOMPP instructions. The only difference is that with the FUCOM, FUCOMP, and FUCOMPP instructions, if an unordered condition is detected because one or both of the operands is a QNaN, the floating-point invalid-operation exception is not generated.

Table 7-15. Setting of FPU Condition Code Flags for Real Number Comparisons

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

The FICOM and FICOMP instructions also operate the same as the FCOM and FCOMP instructions, except that the source operand is an integer value in memory. The integer value is automatically converted into an extended real value prior to making the comparison. The FICOMP instruction pops the FPU register stack following the comparison operation.

The FTST instruction performs the same operation as the FCOM instruction, except that the value in register ST(0) is always compared with the value 0.0.

The FCOMI and FCOMIP instructions are new in the Intel Pentium Pro processor. They perform the same comparison as the FCOM and FCOMP instructions, except that they set the status flags (ZF, PF, and CF) in the EFLAGS register to indicate the results of the comparison (see Table 7-16) instead of the FPU condition code flags. The FCOMI and FCOMIP instructions allow condition branch instructions (J*cc*) to be executed directly from the results of their comparison.

Table 7-16. Setting of EFLAGS Status Flags for Real Number Comparisons

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

The FUCOMI and FUCOMIP instructions operate the same as the FCOMI and FCOMIP instructions, except that they do not generate a floating-point invalid-operation exception if the

unordered condition is the result of one or both of the operands being a QNaN. The FCOMIP and FUCOMIP instructions pop the FPU register stack following the comparison operation.

The FXAM instruction determines the classification of the real value in the ST(0) register (that is, whether the value is zero, a denormal number, a normal finite number, ∞, a NaN, or an unsupported format) or that the register is empty. It sets the FPU condition code flags to indicate the classification (see "FXAM—Examine" in Chapter 3, *Instruction Set Reference*, of the *Intel Architecture Software Developer's Manual, Volume 2*). It also sets the C1 flag to indicate the sign of the value.

7.5.6.1. BRANCHING ON THE FPU CONDITION CODES

The processor does not offer any control-flow instructions that branch on the setting of the condition code flags (C0, C2, and C3) in the FPU status word. To branch on the state of these flags, the FPU status word must first be moved to the AX register in the integer unit. The FSTSW AX (store status word) instruction can be used for this purpose. When these flags are in the AX register, the TEST instruction can be used to control conditional branching as follows:

1. Check for an unordered result. Use the TEST instruction to compare the contents of the AX register with the constant 0400H (see Table 7-17). This operation will clear the ZF flag in the EFLAGS register if the condition code flags indicate an unordered result; otherwise, the ZF flag will be set. The JNZ instruction can then be used to transfer control (if necessary) to a procedure for handling unordered operands.

Table 7-17. TEST Instruction Constants for Conditional Branching

Order	Constant	Branch
ST(0) > Source Operand	4500H	JZ
ST(0) < Source Operand	0100H	JNZ
ST(0) = Source Operand	4000H	JNZ
Unordered	0400H	JNZ

2. Check ordered comparison result. Use the constants given in Table 7-17 in the TEST instruction to test for a less than, equal to, or greater than result, then use the corresponding conditional branch instruction to transfer program control to the appropriate procedure or section of code.

If a program or procedure has been thoroughly tested and it incorporates periodic checks for QNaN results, then it is not necessary to check for the unordered result every time a comparison is made.

See Section 7.3.3., "Branching and Conditional Moves on FPU Condition Codes", for another technique for branching on FPU condition codes.

Some non-comparison FPU instructions update the condition code flags in the FPU status word. To ensure that the status word is not altered inadvertently, store it immediately following a comparison operation.

7.5.7. Trigonometric Instructions

The following instructions perform four common trigonometric functions:

FSIN	Sine
FCOS	Cosine
FSINCOS	Sine and cosine
FPTAN	Tangent
FPATAN	Arctangent

These instructions operate on the top one or two registers of the FPU register stack and they return their results to the stack. The source operands must be given in radians.

The FSINCOS instruction returns both the sine and the cosine of a source operand value. It operates faster than executing the FSIN and FCOS instructions in succession.

The FPATAN instruction computes the arctangent of ST(1) divided by ST(0). It is useful for converting rectangular coordinates to polar coordinates.

7.5.8. Pi

When the argument (source operand) of a trigonometric function is within the range of the function, the argument is automatically reduced by the appropriate multiple of 2π through the same reduction mechanism used by the FPREM and FPREM1 instructions. The internal value of π that the Intel Architecture FPU uses for argument reduction and other computations is as follows:

$\pi = 0.f * 2^2$

where:

f = C90FDAA2 2168C234 C

(The spaces in the fraction above indicate 32-bit boundaries.)

This internal π value has a 66-bit mantissa, which is 2 bits more than is allowed in the significand of an extended-real value. (Since 66 bits is not an even number of hexadecimal digits, two additional zeros have been added to the value so that it can be represented in hexadecimal format. The least-significant hexadecimal digit (C) is thus 1100B, where the two least-significant bits represent bits 67 and 68 of the mantissa.)

This value of π has been chosen to guarantee no loss of significance in a source operand, provided the operand is within the specified range for the instruction.

If the results of computations that explicitly use π are to be used in the FSIN, FCOS, FSINCOS, or FPTAN instructions, the full 66-bit fraction of π should be used. This insures that the results are consistent with the argument-reduction algorithms that these instructions use. Using a rounded version of π can cause inaccuracies in result values, which if propagated through several calculations, might result in meaningless results.

A common method of representing the full 66-bit fraction of π is to separate the value into two numbers (highπ and lowπ) that when added together give the value for π shown earlier in this section with the full 66-bit fraction:

$\pi = \text{high}\pi + \text{low}\pi$

For example, the following two values (given in scientific notation with the fraction in hexadecimal and the exponent in decimal) represent the 33 most-significant and the 33 least-significant bits of the fraction:

highπ (unnormalized)= 0.C90FDAA20 * 2^{+2}

lowπ (unnormalized) = 0.42D184698 * 2^{-31}

These values encoded in standard IEEE double-real format are as follows:

highπ = 400921FB 54400000

lowπ = 3DE0B461 1A600000

(Note that in the IEEE double-real format, the exponents are biased (by 1023) and the fractions are normalized.)

Similar versions of π can also be written in extended-real format.

When using this two-part π value in an algorithm, parallel computations should be performed on each part, with the results kept separate. When all the computations are complete, the two results can be added together to form the final result.

The complications of maintaining a consistent value of π for argument reduction can be avoided, either by applying the trigonometric functions only to arguments within the range of the automatic reduction mechanism, or by performing all argument reductions (down to a magnitude less than π/4) explicitly in software.

7.5.9. Logarithmic, Exponential, and Scale

The following instructions provide two different logarithmic functions, an exponential function, and a scale function.

FYL2X	Compute log ($y * \log_2 x$)
FYL2XP1	Compute log epsilon ($y * \log_2(x + 1)$)
F2XM1	Compute exponential ($2^x - 1$)
FSCALE	Scale

The FYL2X and FYL2XP1 instructions perform two different base 2 logarithmic operations. The FYL2X instruction computes the log of ($y * \log_2 x$). This operation permits the calculation of the log of any base using the following equation:

$\log_b x = (1/\log_2 b) * \log_2 x$

The FYL2XP1 instruction computes the log epsilon of ($y * \log_2 (x + 1)$). This operation provides optimum accuracy for values of epsilon (ε) that are close to 0.

The F2XM1 instruction computes the exponential ($2^x - 1$). This instruction only operates on source values in the range −1.0 to +1.0.

The FSCALE instruction multiplies the source operand by a power of 2.

7.5.10. Transcendental Instruction Accuracy

The algorithms that the Pentium and Pentium Pro processors use for the transcendental instructions (FSIN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, and FYL2XP1) allow a higher level of accuracy than was possible in earlier Intel Architecture math coprocessors and FPUs. The accuracy of these instructions is measured in terms of **units in the last place** (**ulp**). For a given argument *x*, let *f*(*x*) and *F*(*x*) be the correct and computed (approximate) function values, respectively. The error in ulps is defined to be:

$$error = \left| \frac{f(x) - F(x)}{2^{k-63}} \right|$$

where *k* is an integer such that $1 \le 2^{-k} f(x) < 2$.

With the Pentium and Pentium Pro processors, the worst case error on transcendental functions is less than 1 ulp when rounding to the nearest-even and less than 1.5 ulps when rounding in other modes. The functions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

With the Intel486 processor and Intel 387 math coprocessor, the worst-case, transcendental-function error is typically 3 or 3.5 ulps, but is sometimes as large as 4.5 ulps.

7.5.11. FPU Control Instructions

The following instructions control the state and modes of operation of the FPU. They also allow the status of the FPU to be examined:

FINIT/FNINIT	Initialize FPU
FLDCW	Load FPU control word
FSTCW/FNSTCW	Store FPU control word
FSTSW/FNSTSW	Store FPU status word
FCLEX/FNCLEX	Clear FPU exception flags
FLDENV	Load FPU environment
FSTENV/FNSTENV	Store FPU environment
FRSTOR	Restore FPU state
FSAVE/FNSAVE	Save FPU state
FINCSTP	Increment FPU register stack pointer
FDECSTP	Decrement FPU register stack pointer
FFREE	Free FPU register
FNOP	No operation
WAIT/FWAIT	Check for and handle pending unmasked FPU exceptions

The FINIT/FNINIT instructions initialize the FPU and its internal registers to default values.

The FLDCW instructions loads the FPU control word register with a value from memory. The FSTCW/FNSTCW and FSTSW/FNSTSW instructions store the FPU control and status words, respectively, in memory (or for an FSTSW/FNSTSW instruction in a general-purpose register).

The FSTENV/FNSTENV and FSAVE/FNSAVE instructions save the FPU environment and state, respectively, in memory. The FPU environment includes all the FPU's control and status registers; the FPU state includes the FPU environment and the data registers in the FPU register stack. (The FSAVE/FNSAVE instruction also initializes the FPU to default values, like the FINIT/FNINIT instruction, after it saves the original state of the FPU.)

The FLDENV and FRSTOR instructions load the FPU environment and state, respectively, from memory into the FPU. These instructions are commonly used when switching tasks or contexts.

The WAIT/FWAIT instructions are synchronization instructions. (They are actually mnemonics for the same opcode.) These instructions check the FPU status word for pending unmasked FPU exceptions. If any pending unmasked FPU exceptions are found, they are handled before the processor resumes execution of the instructions (integer, floating-point, or system instruction) in the instruction stream. The WAIT/FWAIT instructions are provided to allow synchronization of instruction execution between the FPU and the processor's integer unit. See Section 7.9., "Floating-Point Exception Synchronization" for more information on the use of the WAIT/FWAIT instructions.

7.5.12. Waiting Vs. Non-waiting Instructions

All of the floating-point instructions except a few special control instructions perform a wait operation (similar to the WAIT/FWAIT instructions), to check for and handle pending unmasked FPU exceptions, before they perform their primary operation (such as adding two real numbers). These instructions are called **waiting** instructions. Some of the FPU control instructions, such as FSTSW/FNSTSW, have both a waiting and a non-waiting versions. The waiting version (with the "F" prefix) executes a wait operation before it performs its primary operation; whereas, the non-waiting version (with the "FN" prefix) ignores pending unmasked exceptions. Non-waiting instructions allow software to save the current FPU state without first handling pending exceptions or to reset or reinitialize the FPU without regard for pending exceptions.

NOTE

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for a non-waiting instruction to be interrupted prior to being executed to handle a pending FPU exception. The circumstances where this can happen and the resulting action of the processor are described in Section D.2.1.3., "No-Wait FPU Instructions Can Get FPU Interrupt in Window". When operating a Pentium Pro processor in MS-DOS compatibility mode, non-waiting instructions can not be interrupted in this way (see Section D.2.2., "MS-DOS* Compatibility Mode in the Pentium® Pro Processor").

7.5.13. Unsupported FPU Instructions

The Intel 8087 instructions FENI and FDISI and the Intel 287 math coprocessor instruction FSETPM perform no function in the Intel 387 math coprocessor, or the Intel486, Pentium, or Pentium Pro processors. If these opcodes are detected in the instruction stream, the FPU performs no specific operation and no internal FPU states are affected.

7.6. OPERATING ON NANS

As was described in Section 7.2.3.4., "NaNs", the FPU supports two types of NaNs: SNaNs and QNaNs. An SNaN is any NaN value with its most-significant fraction bit set to 0 and at least one other fraction bit set to 1. (If all the fraction bits are set to 0, the value is an ∞.) A QNaN is any NaN value with the most-significant fraction bit set to 1. The sign bit of a NaN is not interpreted.

As a general rule, when a QNaN is used in one or more arithmetic floating-point instructions, it is allowed to propagate through a computation. An SNaN on the other hand causes a floating-point invalid-operation exception to be signaled. SNaNs are typically used to trap or invoke an exception handler. They must be inserted by software; that is, the FPU never generates an SNaN as a result.

The floating-point invalid-operation exception has a flag and a mask bit associated with it in the FPU status and control registers, respectively (see Section 7.7., "Floating-Point Exception Handling"). The mask bit determines how the FPU handles an SNaN value. If the floating-point invalid-operation mask bit is set, the SNaN is converted to a QNaN by setting the most-significant fraction bit of the value to 1. The result is then stored in the destination operand and the floating-point invalid-operation flag is set. If the invalid-operation mask is clear, a floating-point invalid-operation fault is signaled and no result is stored in the destination operand.

When a real operation or exception delivers a QNaN result, the value of the result depends on the source operands, as shown in Table 7-18.

Except for the rules given at the beginning of this section for encoding SNaNs and QNaNs, software is free to use the bits in the significand of a NaN for any purpose. Both SNaNs and QNaNs can be encoded to carry and store data, such as diagnostic information.

Table 7-18. Rules for Generating QNaNs

Source Operands	QNaN Result
An SNaN and a QNaN.	The QNaN source operand.
Two SNaNs.	The SNaN with the larger significand converted into a QNaN.
Two QNaNs.	The QNaN with the larger significand.
An SNaN and a real value.	The SNaN converted into a QNaN.
A QNaN and a real value.	The QNaN source operand.
Neither source operand is a NaN and a floating-point invalid-operation exception is signaled.	The default QNaN *real indefinite*.

7.6.1. Uses for Signaling NANs

By unmasking the invalid operation exception, the programmer can use signaling NaNs to trap to the exception handler. The generality of this approach and the large number of NaN values that are available provide the sophisticated programmer with a tool that can be applied to a variety of special situations.

For example, a compiler can use signaling NaNs as references to uninitialized (real) array elements. The compiler can preinitialize each array element with a signaling NaN whose significand contained the index (relative position) of the element. Then, if an application program attempts to access an element that it had not initialized, it can use the NaN placed there by the compiler. If the invalid operation exception is unmasked, an interrupt will occur, and the exception handler will be invoked. The exception handler can determine which element has been accessed, since the operand address field of the exception pointers will point to the NaN, and the NaN will contain the index number of the array element.

7.6.2. Uses for Quiet NANs

Quiet NaNs are often used to speed up debugging. In its early testing phase, a program often contains multiple errors. An exception handler can be written to save diagnostic information in memory whenever it was invoked. After storing the diagnostic data, it can supply a quiet NaN as the result of the erroneous instruction, and that NaN can point to its associated diagnostic area in memory. The program will then continue, creating a different NaN for each error. When the program ends, the NaN results can be used to access the diagnostic data saved at the time the errors occurred. Many errors can thus be diagnosed and corrected in one test run.

In embedded applications which use computed results in further computations, an undetected QNaN can invalidate all subsequent results. Such applications should therefore periodically check for QNaNs and provide a recovery mechanism to be used if a QNaN result is detected.

7.7. FLOATING-POINT EXCEPTION HANDLING

The FPU detects six classes of exception conditions while executing floating-point instructions:

- Invalid operation (#I)
  - — Stack overflow or underflow (#IS)
  - — Invalid arithmetic operation (#IA)
- Divide-by-zero (#Z)
- Denormalized operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

The nomenclature of "#" symbol followed by one or two letters (for example, #IS) is used in this manual to indicate exception conditions. It is merely a short-hand form and is not related to assembler mnemonics.

Each of the six exception classes has a corresponding flag bit in the FPU status word and a mask bit in the FPU control word (see Section 7.3.2., "FPU Status Register" and Section 7.3.4., "FPU Control Word", respectively). In addition, the exception summary (ES) flag in the status word indicates when any of the exceptions has been detected, and the stack fault (SF) flag (also in the status word) distinguishes between the two types of invalid-operation exceptions.

When the FPU detects a floating-point exception, it sets the appropriate flags in the FPU status word, then takes one of two possible courses of action:

- Handles the exception automatically, producing a predefined (and often times usable result), while allowing program execution to continue undisturbed.
- Invokes a software exception handler to handle the exception.

The following sections describe how the FPU handles exceptions (either automatically or by calling a software exception handler), how the FPU detects the various floating-point exceptions, and the automatic (masked) response to the floating-point exceptions.

7.7.1. Arithmetic vs. Non-arithmetic Instructions

When dealing with floating-point exceptions, it is useful to distinguish between **arithmetic instructions** and **non-arithmetic instructions**. Non-arithmetic instructions have no operands or do not make substantial changes to their operands. Arithmetic instructions do make significant changes to their operands; in particular, they make changes that could result in a floating-point exception being signaled. Table 7-19 lists the non-arithmetic and arithmetic instructions. It should be noted that some non-arithmetic instructions can signal a floating-point stack (fault) exception, but this exception is not the result of an operation on an operand.

7.7.2. Automatic Exception Handling

If the FPU detects an exception condition for a masked exception (an exception with its mask bit set), it sets the exception flag for the exception and delivers a predefined (default) response and continues executing instructions. The masked (default) responses to exceptions have been chosen to deliver a reasonable result for each exception condition and are generally satisfactory for most floating-point applications. By masking or unmasking specific floating-point exceptions in the FPU control word, programmers can delegate responsibility for most exceptions to the FPU and reserve the most severe exception conditions for software exception handlers.

Because the exception flags are "sticky," they provide a cumulative record of the exceptions that have occurred since they were last cleared. A programmer can thus mask all exceptions, run a calculation, and then inspect the exception flags to see if any exceptions were detected during the calculation.

Table 7-19. Arithmetic and Non-arithmetic Instructions

Non-arithmetic Instructions	Arithmetic Instructions
FABS	F2XM1
FCHS	FADD/FADDP
FCLEX	FBLD
FDECSTP	FBSTP
FFREE	FCOM/FCOMP/FCOMPP
FINCSTP	FCOS
FINIT/FNINIT	FDIV/FDIVP/FDIVR/FDIVRP
FLD (register-to-register)	FIADD
FLD (extended format from memory)	FICOM/FICOMP
FLD constant	FIDIV/FIDIVR
FLDCW	FILD
FLDENV	FIMUL
FNOP	FIST/FISTP
FRSTOR	FISUB/FISUBR
FSAVE/FNSAVE	FLD (conversion)
FST/FSTP (register-to-register)	FMUL/FMULP
FSTP (extended format to memory)	FPATAN
FSTCW/FNSTCW	FPREM/FPREM1
FSTENV/FNSTENV	FPTAN
FSTSW/FNSTSW	FRNDINT
WAIT/FWAIT	FSCALE
FXAM	FSIN
FXCH	FSINCOS
	FSQRT
	FST/FSTP (conversion)
	FSUB/FSUBP/FSUBR/FSUBRP
	FTST
	FUCOM/FUCOMP/FUCOMPP
	FXTRACT
	FYL2X/FYL2XP1

Note that when exceptions are masked, the FPU may detect multiple exceptions in a single instruction, because it continues executing the instruction after performing its masked response. For example, the FPU can detect a denormalized operand, perform its masked response to this exception, and then detect numeric underflow.

7.7.3. Software Exception Handling

The FPU in the Pentium Pro, Pentium, and Intel486 processors provides two different modes of operation for invoking a software exception handler for floating-point exceptions: native mode and MS-DOS compatibility mode. The mode of operation is selected with the NE flag of control register CR0. (See Chapter 2, *System Architecture Overview*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information about the NE flag.)

7.7.3.1. NATIVE MODE

The native mode for handling floating-point exceptions is selected by setting the NE flag in control register CR0 to 1. In this mode, if the FPU detects an exception condition while executing a floating-point instruction and the exception is unmasked (the mask bit for the exception is cleared), the FPU sets the flag for the exception and the ES flag in the FPU status word. It then invokes the software exception handler through the floating-point-error exception (#MF, vector 16), immediately before execution of any of the following instructions in the processor's instruction stream:

- The next floating-point instruction, unless it is one of the non-waiting instructions (FNINIT, FNCLEX, FNSTSW, FNSTCW, FNSTENV, and FNSAVE).
- The next WAIT/FWAIT instruction.
- The next MMX instruction.

If the next floating-point instruction in the instruction stream is a non-waiting instruction, the FPU executes the instruction without invoking the software exception handler.

7.7.3.2. MS-DOS* COMPATIBILITY MODE

If the NE flag in control register CR0 is set to 0, the MS-DOS compatibility mode for handling floating-point exceptions is selected. In this mode, the software exception handler for floating-point exceptions is invoked externally using the processor's FERR#, INTR, and IGNNE# pins. This method of reporting floating-point errors and invoking an exception handler is provided to support the floating-point exception handling mechanism used in PC systems that are running the MS-DOS or Windows* 95 operating system.

The MS-DOS compatibility mode is typically used as follows to invoke the floating-point exception handler:

1. If the FPU detects an unmasked floating-point exception, it sets the flag for the exception and the ES flag in the FPU status word.

2. If the IGNNE# pin is deasserted, the FPU then asserts the FERR# pin either immediately, or else delayed (deferred) until just before the execution of the next waiting floating-point instruction or MMX™ instruction. Whether the FERR# pin is asserted immediately or delayed depends on the type of processor, the instruction, and the type of exception.
3. If a preceding floating-point instruction has set the exception flag for an unmasked FPU exception, the processor freezes just before executing the **next** WAIT instruction, waiting floating-point instruction, or MMX instruction. Whether the FERR# pin was asserted at the preceding floating-point instruction or is just now being asserted, the freezing of the processor assures that the FPU exception handler will be invoked before the new floating-point (or MMX) instruction gets executed.
4. The FERR# pin is connected through external hardware to IRQ13 of a cascaded, programmable interrupt controller (PIC). When the FERR# pin is asserted, the PIC is programmed to generate an interrupt 75H.
5. The PIC asserts the INTR pin on the processor to signal the interrupt 75H.
6. The BIOS for the PC system handles the interrupt 75H by branching to the interrupt 2 (NMI) interrupt handler.
7. The interrupt 2 handler determines if the interrupt is the result of an NMI interrupt or a floating-point exception.
8. If a floating-point exception is detected, the interrupt 2 handler branches to the floating-point exception handler.

If the IGNNE# pin is asserted, the processor ignores floating-point error conditions. This pin is provided to inhibit floating-point exceptions from being generated while the floating-point exception handler is servicing a previously signaled floating-point exception.

Appendix D, *Guidelines for Writing FPU Exception Handlers*, describes the MS-DOS compatibility mode in much greater detail. This mode is somewhat more complicated in the Intel486 and Pentium processor implementations, as described in Appendix D.

7.7.3.3. TYPICAL FLOATING-POINT EXCEPTION HANDLER ACTIONS

After the floating-point exception handler is invoked, the processor handles the exception in the same manner that it handles non-FPU exceptions. (The floating-point exception handler is normally part of the operating system or executive software.) A typical action of the exception handler is to store FPU state information in memory (with the FSTENV/FNSTENV or FSAVE/FNSAVE instructions) so that it can evaluate the exception and formulate an appropriate response (see Section 7.3.9., “Saving the FPU’s State”). Other typical exception handler actions include:

- Examining stored FPU state information (control, status, and tag words, and FPU instruction and operand pointers) to determine the nature of the error.
- Correcting the condition that caused the error.
- Clearing the exception bits in the status word.
- Returning to the interrupted program and resuming normal execution.

If the faulting floating-point instruction is followed by one or more non-floating-point instructions, it may not be useful to re-execute the faulting instruction. See Section 7.9., "Floating-Point Exception Synchronization", for more information on synchronizing floating-point exceptions.

In cases where the handler needs to restart program execution with the faulting instruction, the IRET instruction cannot be used directly. The reason for this is that because the exception is not generated until the next floating-point or WAIT/FWAIT instruction following the faulting floating-point instruction, the return instruction pointer on the stack may not point to the faulting instruction. To restart program execution at the faulting instruction, the exception handler must obtain a pointer to the instruction from the saved FPU state information, load it into the return instruction pointer location on the stack, and then execute the IRET instruction.

In lieu of writing recovery procedures, the exception handler can do the following:

- Increment an exception counter for later display or printing.
- Print or display diagnostic information (such as, the FPU environment and registers).
- Halt further program execution.

See Section D.3.4., "FPU Exception Handling Examples", for general examples of floating-point exception handlers and for specific examples of how to write a floating-point exception handler when using the MS-DOS compatibility mode.

7.8. FLOATING-POINT EXCEPTION CONDITIONS

The following sections describe the various conditions that cause a floating-point exception to be generated and the masked response of the FPU when these conditions are detected. Chapter 3, *Instruction Set Reference*, in the *Intel Architecture Software Developer's Manual, Volume 2*, lists the floating-point exceptions that can be signaled for each floating-point instruction.

7.8.1. Invalid Operation Exception

The floating-point invalid-operation exception occurs in response to two general types of operations:

- Stack overflow or underflow (#IS).
- Invalid arithmetic operand (#IA).

The flag for this exception (IE) is bit 0 of the FPU status word, and the mask bit (IM) is bit 0 of the FPU control word. The stack fault flag (SF) of the FPU status word indicates the type of operation caused the exception. When the SF flag is set to 1, a stack operation has resulted in stack overflow or underflow; when the flag is cleared to 0, an arithmetic instruction has encountered an invalid operand. Note that the FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition. As a result, the state of the SF flag can be 1 following an invalid-arithmetic-operation exception, if it was not cleared from the last time a stack overflow or underflow condition occurred. See Section 7.3.2.4., "Stack Fault Flag", for more information about the SF flag.

7.8.1.1. STACK OVERFLOW OR UNDERFLOW EXCEPTION (#IS)

The FPU tag word keeps track of the contents of the registers in the FPU register stack (see Section 7.3.6., "FPU Tag Word"). It then uses this information to detect two different types of stack faults:

- Stack overflow—an instruction attempts to write a value into a non-empty FPU register
- Stack underflow—an instruction attempts to read a value from an empty FPU register.

When the FPU detects stack overflow or underflow, it sets the IE flag (bit 0) and the SF flag (bit 6) in the FPU status word to 1. It then sets condition-code flag C1 (bit 9) in the FPU status word to 1 if stack overflow occurred or to 0 if stack underflow occurred.

If the invalid-operation exception is masked, the FPU then returns the real, integer, or BCD-integer indefinite value to the destination operand, depending on the instruction being executed. This value overwrites the destination register or memory location specified by the instruction.

If the invalid-operation exception is not masked, a software exception handler is invoked (see Section 7.7.3., "Software Exception Handling") and the top-of-stack pointer (TOP) and source operands remain unchanged.

The term stack overflow comes from the condition where the a program has pushed eight values onto the FPU register stack and the next value pushed on the stack causes a stack wraparound to a register that already contains a value. The term stack underflow refers to the opposite condition from stack overflow. Here, a program has popped eight values from the FPU register stack and the next value popped from the stack causes stack wraparound to an empty register.

7.8.1.2. INVALID ARITHMETIC OPERAND EXCEPTION (#IA)

The FPU is able to detect a variety of invalid arithmetic operations that can be coded in a program. These operations generally indicate a programming error, such as dividing ∞ by ∞. Table 7-20 lists the invalid arithmetic operations that the FPU detects. This group includes the invalid operations defined in IEEE Std. 854.

When the FPU detects an invalid arithmetic operand, it sets the IE flag (bit 0) in the FPU status word to 1. If the invalid-operation exception is masked, the FPU then returns an indefinite value to the destination operand or sets the floating-point condition codes, as shown in Table 7-20. If the invalid-operation exception is not masked, a software exception handler is invoked (see Section 7.7.3., "Software Exception Handling") and the top-of-stack pointer (TOP) and source operands remain unchanged.

Table 7-20. Invalid Arithmetic Operations and the Masked Responses to Them

Condition	Masked Response
Any arithmetic operation on an operand that is in an unsupported format.	Return the real indefinite value to the destination operand.
Any arithmetic operation on a SNaN.	Return a QNaN to the destination operand (see Section 7.6., “Operating on NaNs”).
Compare and test operations: one or both operands are NaNs.	Set the condition code flags (C0, C2, and C3) in the FPU status word to 111B (not comparable).
Addition: operands are opposite-signed infinities. Subtraction: operands are like-signed infinities.	Return the real indefinite value to the destination operand.
Multiplication: ∞ by 0; 0 by ∞.	Return the real indefinite value to the destination operand.
Division: ∞ by ∞; 0 by 0.	Return the real indefinite value to the destination operand.
Remainder instructions FPREM, FPREM1: modulus (divisor) is 0 or dividend is ∞.	Return the real indefinite; clear condition code flag C2 to 0.
Trigonometric instructions FCOS, FPTAN, FSIN, FSINCOS: source operand is ∞.	Return the real indefinite; clear condition code flag C2 to 0.
FSQRT: negative operand (except FSQRT (–0) = –0); FYL2X: negative operand (except FYL2X (–0) = –∞); FYL2XP1: operand more negative than –1.	Return the real indefinite value to the destination operand.
FBSTP: source register is empty or it contains a NaN, ∞, or a value that cannot be represented in 18 decimal digits.	Store BCD integer indefinite value in the destination operand.
FXCH: one or both registers are tagged empty.	Load empty registers with the real indefinite value, then perform the exchange.

7.8.2. Divide-By-Zero Exception (#Z)

The FPU reports a floating-point zero-divide exception whenever an instruction attempts to divide a finite non-zero operand by 0. The flag (ZE) for this exception is bit 2 of the FPU status word, and the mask bit (ZM) is bit 2 of the FPU control word. The FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, and FIDIVR instructions and the other instructions that perform division internally (FYL2X and FXTRACT) can report the divide-by-zero exception.

When a divide-by-zero exception occurs and the exception is masked, the FPU sets the ZE flag and returns the values shown in Table 7-21. If the divide-by-zero exception is not masked, the ZE flag is set, a software exception handler is invoked (see Section 7.7.3., “Software Exception Handling”), and the top-of-stack pointer (TOP) and source operands remain unchanged.

Table 7-21. Divide-By-Zero Conditions and the Masked Responses to Them

Condition	Masked Response
Divide or reverse divide operation with a 0 divisor.	Returns an ∞ signed with the exclusive OR of the sign of the two operands to the destination operand.
FYL2X instruction.	Returns an ∞ signed with the opposite sign of the non-zero operand to the destination operand.
FXTRACT instruction.	ST(1) is set to –∞; ST(0) is set to 0 with the same sign as the source operand.

7.8.3. Denormal Operand Exception (#D)

The FPU signals the denormal-operand exception under the following conditions:

- If an arithmetic instruction attempts to operate on a denormal operand (see Section 7.2.3.2., "Normalized and Denormalized Finite Numbers").
- If an attempt is made to load a denormal single- or double-real value into an FPU register. (If the denormal value being loaded is an extended-real value, the denormal-operand exception is not reported.)

The flag (DE) for this exception is bit 1 of the FPU status word, and the mask bit (DM) is bit 1 of the FPU control word.

When a denormal-operand exception occurs and the exception is masked, the FPU sets the DE flag, then proceeds with the instruction. The denormal operand in single- or double-real format is automatically normalized when converted to the extended-real format. Operating on denormal numbers will produce results at least as good as, and often better than, what can be obtained when denormal numbers are flushed to zero. In fact, subsequent operations will benefit from the additional precision of the internal extended-real format. Most programmers mask this exception so that a computation may proceed, then analyze any loss of accuracy when the final result is delivered.

When a denormal-operand exception occurs and the exception is not masked, the DE flag is set and a software exception handler is invoked (see Section 7.7.3., "Software Exception Handling"). The top-of-stack pointer (TOP) and source operands remain unchanged. When denormal operands have reduced significance due to loss of low-order bits, it may be advisable to not operate on them. Precluding denormal operands from computations can be accomplished by an exception handler that responds to unmasked denormal-operand exceptions.

7.8.4. Numeric Overflow Exception (#O)

The FPU reports a floating-point numeric overflow exception (#O) whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that will fit into the real format of the destination operand. For example, if the destination format is extended-real (80 bits), overflow occurs when the rounded result falls outside the unbiased range of $-1.0 * 2^{16384}$ to $1.0 * 2^{16384}$ (exclusive). Numeric overflow can occur on arithmetic operations where the result is stored in an FPU data register. It can also occur on store-real operations (with the FST and

FSTP instructions), where a within-range value in a data register is stored in memory in a single- or double-real format. The overflow threshold range for the single-real format is $-1.0 * 2^{128}$ to $1.0 * 2^{128}$; the range for the double-real format is $-1.0 * 2^{1024}$ to $1.0 * 2^{1024}$.

The numeric overflow exception cannot occur when overflow occurs when storing values in an integer or BCD integer format. Instead, the invalid-arithmetic-operand exception is signaled.

The flag (OE) for the numeric-overflow exception is bit 3 of the FPU status word, and the mask bit (OM) is bit 3 of the FPU control word.

When a numeric-overflow exception occurs and the exception is masked, the FPU sets the OE flag and returns one of the values shown in Table 7-22. The value returned depends on the current rounding mode of the FPU (see Section 7.3.4.3., "Rounding Control Field").

Table 7-22. Masked Responses to Numeric Overflow

Rounding Mode	Sign of True Result	Result
To nearest	+	+∞
	–	–∞
Toward –∞	+	Largest finite positive number
	–	–∞
Toward +∞	+	+∞
	–	Largest finite negative number
Toward zero	+	Largest finite positive number
	–	Largest finite negative number

The action that the FPU takes when numeric overflow occurs and the numeric-overflow exception is not masked, depends on whether the instruction is supposed to store the result in memory or on the register stack.

If the destination is a memory location, the OE flag is set and a software exception handler is invoked (see Section 7.7.3., "Software Exception Handling"). The top-of-stack pointer (TOP) and source and destination operands remain unchanged.

If the destination is the register stack, the exponent of the rounded result is divided by 2^{24576} and the result is stored along with the significand in the destination operand. Condition code bit C1 in the FPU status word (called in this situation the "round-up bit") is set if the significand was rounded upward and cleared if the result was rounded toward 0. After the result is stored, the OE flag is set and a software exception handler is invoked.

The scaling bias value 24,576 is equal to $3 * 2^{13}$. Biasing the exponent by 24,576 normally translates the number as nearly as possible to the middle of the extended-real exponent range so that, if desired, it can be used in subsequent scaled operations with less risk of causing further exceptions.

When using the FSCALE instruction, massive overflow can occur, where the result is too large to be represented, even with a bias-adjusted exponent. Here, if overflow occurs again, after the result has been biased, a properly signed ∞ is stored in the destination operand.

7.8.5. Numeric Underflow Exception (#U)

The FPU reports a floating-point numeric underflow exception (#U) whenever the rounded result of an arithmetic instruction is "tiny" (that is, less than the smallest possible normalized, finite value that will fit into the real format of the destination operand). For example, if the destination format is extended-real (80 bits), underflow occurs when the rounded result falls in the unbiased range of $-1.0 * 2^{-16382}$ to $1.0 * 2^{-16382}$ (exclusive). Like numeric overflow, numeric underflow can occur on arithmetic operations where the result is stored in an FPU data register. It can also occur on store-real operations (with the FST and FSTP instructions), where a within-range value in a data register is stored in memory in a single- or double-real format. The underflow threshold range for the single-real format is $-1.0 * 2^{-126}$ to $1.0 * 2^{-126}$; the range for the double-real format is $-1.0 * 2^{-1022}$ to $1.0 * 2^{-1022}$. (The numeric underflow exception cannot occur when storing values in an integer or BCD integer format.)

The flag (UE) for the numeric-underflow exception is bit 4 of the FPU status word, and the mask bit (UM) is bit 4 of the FPU control word.

When a numeric-underflow exception occurs and the exception is masked, the FPU denormalizes the result (see Section 7.2.3.2., "Normalized and Denormalized Finite Numbers"). If the denormalized result is exact, the FPU stores the result in the destination operand, without setting the UE flag. If the denormal result is inexact, the FPU sets the UE flag, then goes on to handle the inexact-result exception condition (see Section 7.8.6., "Inexact-Result (Precision) Exception (#P)"). It is important to note that if numeric-underflow is masked, a numeric-underflow exception is signaled only if the denormalized result is inexact. If the denormalized result is exact, no flags are set and no exceptions are signaled.

The action that the FPU takes when numeric underflow occurs and the numeric-underflow exception is not masked, depends on whether the instruction is supposed to store the result in memory or on the register stack.

If the destination is a memory location, the UE flag is set and a software exception handler is invoked (see Section 7.7.3., "Software Exception Handling"). The top-of-stack pointer (TOP) and source and destination operands remain unchanged.

If the destination is the register stack, the exponent of the rounded result is multiplied by 2^{24576} and the product is stored along with the significand in the destination operand. Condition code bit C1 in the FPU the status register (acting here as a "round-up bit") is set if the significand was rounded upward and cleared if the result was rounded toward 0. After the result is stored, the UE flag is set and a software exception handler is invoked.

The scaling bias value 24,576 is the same as is used for the overflow exception and has the same effect, which is to translate the result as nearly as possible to the middle of the extended-real exponent range.

When using the FSCALE instruction, massive underflow can occur, where the result is too tiny to be represented, even with a bias-adjusted exponent. Here, if underflow occurs again, after the result has been biased, a properly signed 0 is stored in the destination operand.

7.8.6. Inexact-Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary form. This exception occurs frequently and indicates that some (normally acceptable) accuracy has been lost. The exception is supported for applications that need to perform exact arithmetic only. Because the rounded result is generally satisfactory for most applications, this exception is commonly masked. Note that the transcendental instructions [FSIN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, and FYL2XP1] by nature produce inexact results.

The inexact-result exception flag (PE) is bit 5 of the FPU status word, and the mask bit (PM) is bit 5 of the FPU control word.

If the inexact-result exception is masked when an inexact-result condition occurs and a numeric overflow or underflow condition has not occurred, the FPU sets the PE flag and stores the rounded result in the destination operand. The current rounding mode determines the method used to round the result (see Section 7.3.4.3., "Rounding Control Field"). The C1 (round-up) bit in the FPU status word indicates whether the inexact result was rounded up (C1 is set) or "not rounded up" (C1 is cleared). In the "not rounded up" case, the least-significant bits of the inexact result are truncated so that the result fits in the destination format.

If the inexact-result exception is not masked when an inexact result occurs and numeric overflow or underflow has not occurred, the FPU performs the same operation described in the previous paragraph and, in addition, invokes a software exception handler (see Section 7.7.3., "Software Exception Handling").

If an inexact result occurs in conjunction with numeric overflow or underflow, one of the following operations is carried out:

- If an inexact result occurs along with masked overflow or underflow, the OE or UE flag and the PE flag are set and the result is stored as described for the overflow or underflow exceptions (see Section 7.8.4., "Numeric Overflow Exception (#O)" or Section 7.8.5., "Numeric Underflow Exception (#U)"). If the inexact-result exception is unmasked, the FPU also invokes the software exception handler.
- If an inexact result occurs along with unmasked overflow or underflow and the destination operand is a register, the OE or UE flag and the PE flag are set, the result is stored as described for the overflow or underflow exceptions, and the software exception handler is invoked.
- If an inexact result occurs along with unmasked overflow or underflow and the destination operand is a memory location, the inexact-result condition is ignored.

7.8.7. Exception Priority

The processor handles exceptions according to a predetermined precedence. When an instruction generates two or more exception conditions, the exception precedence sometimes results in the higher-priority exception being handled and the lower-priority exceptions being ignored. For

example, dividing an SNaN by zero can potentially signal an invalid-arithmetic-operand exception (due to the SNaN operand) and a divide-by-zero exception. Here, if both exceptions are masked, the FPU handles the higher-priority exception only (the invalid-arithmetic-operand exception), returning a real indefinite to the destination. Alternately, a denormal-operand or inexact-result exception can accompany a numeric underflow or overflow exception, with both exceptions being handled.

The precedence for floating-point exceptions is as follows:

1. Invalid-operation exception, subdivided as follows:
   a. Stack underflow.
   b. Stack overflow.
   c. Operand of unsupported format.
   d. SNaN operand.
2. QNaN operand. Though this is not an exception, the handling of a QNaN operand has precedence over lower-priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a zero-divide exception.
3. Any other invalid-operation exception not mentioned above or a divide-by-zero exception.
4. Denormal-operand exception. If masked, then instruction execution continues, and a lower-priority exception can occur as well.
5. Numeric overflow and underflow exceptions in conjunction with the inexact-result exception.
6. Inexact-result exception.

Invalid operation, zero divide, and denormal operand exceptions are detected before a floating-point operation begins, whereas overflow, underflow, and precision errors are not detected until a true result has been computed. When a **pre-operation** exception is detected, the FPU register stack and memory have not yet been updated, and appear as if the offending instructions has not been executed. When a **post-operation** exception is detected, the register stack and memory may be updated with a result (depending on the nature of the error).

7.9. FLOATING-POINT EXCEPTION SYNCHRONIZATION

Because the integer unit and FPU are separate execution units, it is possible for the processor to execute floating-point, integer, and system instructions concurrently. No special programming techniques are required to gain the advantages of concurrent execution. (Floating-point instructions are placed in the instruction stream along with the integer and system instructions.) However, concurrent execution can cause problems for floating-point exception handlers.

This problem is related to the way the FPU signals the existence of unmasked floating-point exceptions. (Special exception synchronization is not required for masked floating-point exceptions, because the FPU always returns a masked result to the destination operand.)

When a floating-point exception is unmasked and the exception condition occurs, the FPU stops further execution of the floating-point instruction and signals the exception event. On the next occurrence of a floating-point instruction or a WAIT/FWAIT instruction in the instruction stream, the processor checks the ES flag in the FPU status word for pending floating-point exceptions. It floating-point exceptions are pending, the FPU makes an implicit call (traps) to the floating-point software exception handler. The exception handler can then execute recovery procedures for selected or all floating-point exceptions.

Synchronization problems occur in the time frame between when the exception is signaled and when it is actually handled. Because of concurrent execution, integer or system instructions can be executed during this time frame. It is thus possible for the source or destination operands for a floating-point instruction that faulted to be overwritten in memory, making it impossible for the exception handler to analyze or recover from the exception.

To solve this problem, an exception synchronizing instruction (either a floating-point instruction or a WAIT/FWAIT instruction) can be placed immediately after any floating-point instruction that might present a situation where state information pertaining to a floating-point exception might be lost or corrupted. Floating-point instructions that store data in memory are prime candidates for synchronization. For example, the following three lines of code have the potential for exception synchronization problems:

```
FILD COUNT ; Floating-point instruction
INC COUNT  ; Integer instruction
FSQRT      ; Subsequent floating-point instruction
```

In this example, the INC instruction modifies the result of a floating-point instruction (FILD). If an exception is signaled during the execution of the FILD instruction, the result stored in the COUNT memory location might be overwritten before the exception handler is called.

Rearranging the instructions, as follows, so that the FSQRT instruction follows the FILD instruction, synchronizes the exception handling and eliminates the possibility of the exception being handled incorrectly.

```
FILD COUNT ; Floating-point instruction
FSQRT      ; Subsequent floating-point instruction synchronizes
           ; any exceptions generated by the FILD instruction.
INC COUNT  ; Integer instruction
```

The FSQRT instruction does not require any synchronization, because the results of this instruction are stored in the FPU data registers and will remain there, undisturbed, until the next floating-point or WAIT/FWAIT instruction is executed. To absolutely insure that any exceptions emanating from the FSQRT instruction are handled (for example, prior to a procedure call), a WAIT instruction can be placed directly after the FSQRT instruction.

Note that some floating-point instructions (non-waiting instructions) do not check for pending unmasked exceptions (see Section 7.5.11., "FPU Control Instructions"). They include the FNINIT, FNSTENV, FNSAVE, FNSTSW, FNSTCW, and FNCLEX instructions. When an FNINIT, FNSTENV, FNSAVE, or FNCLEX instruction is executed, all pending exceptions are essentially lost (either the FPU status register is cleared or all exceptions are masked). The FNSTSW and FNSTCW instructions do not check for pending interrupts, but they do not modify the FPU status and control registers. A subsequent "waiting" floating-point instruction can then handle any pending exceptions.

intel®

8

Programming With the Intel MMX™ Technology

intel®

CHAPTER 8
PROGRAMMING WITH THE INTEL MMX™ TECHNOLOGY

The Intel MMX technology comprises a set of extensions to the Intel Architecture that are designed to greatly enhance the performance of advanced media and communications applications. These extensions (which include new registers, data types, and instructions) are combined with a single-instruction, multiple-data (SIMD) execution model to accelerate the performance of applications such as motion video, combined graphics with video, image processing, audio synthesis, speech synthesis and compression, telephony, video conferencing, and 2D and 3D graphics, which typically use compute-intensive algorithms to perform repetitive operations on large arrays of simple, native data elements.

The MMX technology defines a simple and flexible software model, with no new mode or operating-system visible state. All existing software will continue to run correctly, without modification, on Intel Architecture processors that incorporate the MMX technology, even in the presence of existing and new applications that incorporate this technology.

The following sections of this chapter describe the MMX technology's basic programming environment, including the MMX register set, data types, and instruction set. Detailed descriptions of the MMX instructions are provided in Chapter 3, *Instruction Set Reference*, of the *Intel Architecture Software Developer's Manual, Volume 2*. The manner in which the MMX technology is integrated into the Intel Architecture system programming model is described in Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*.

8.1. OVERVIEW OF THE MMX™ TECHNOLOGY PROGRAMMING ENVIRONMENT

MMX technology provides the following new extensions to the Intel Architecture programming environment.

- Eight MMX™ registers (MM0 through MM7).
- Four MMX data types (packed bytes, packed words, packed doublewords, and quadword).
- The MMX instruction set.

The MMX registers and data types are described in the following sections. See Section 8.3., "Overview of the MMX™ Instruction Set", for an overview of the MMX instructions.

8.1.1. MMX™ Registers

The MMX register set consists of eight 64-bit registers (see Figure 8-1). The MMX instructions access the MMX registers directly using the register names MM0 through MM7. These registers can only be used to perform calculations on MMX data types; they cannot be used to address memory. Addressing of MMX instruction operands in memory is handled by using the standard Intel Architecture addressing modes and general-purpose registers (EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP).

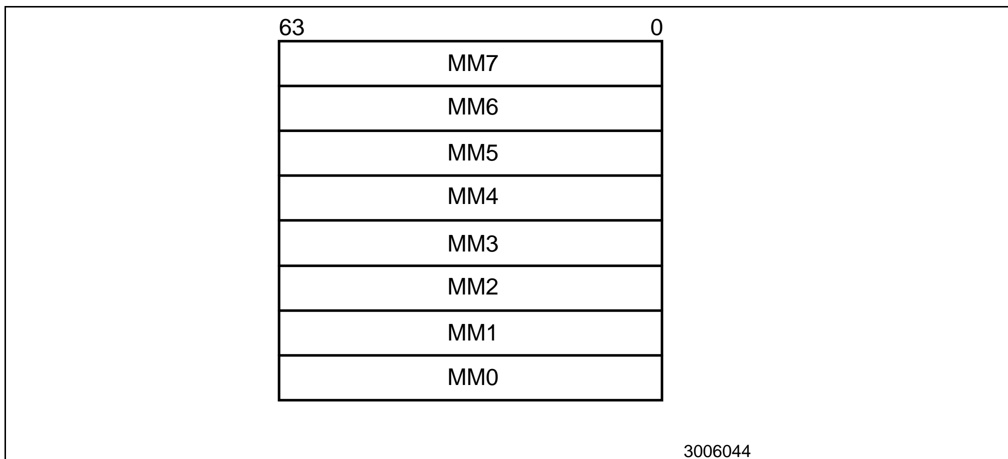


Figure 8-1. MMX™ Register Set

Although the MMX registers are defined in the Intel Architecture as separate registers, they are aliased to the registers in the FPU data register stack (R0 through R7). (See Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more a detailed discussion of the aliasing of MMX registers.)

8.1.2. MMX™ Data Types

The MMX technology defines the following new 64-bit data types (see Figure 8-2):

Packed bytes	Eight bytes packed into one 64-bit quantity.
Packed words	Four (16-bit) words packed into one 64-bit quantity.
Packed doublewords	Two (32-bit) doublewords packed into one 64-bit quantity.
Quadword	One 64-bit quantity.

The bytes in the packed bytes data type are numbered 0 through 7, with byte 0 being contained in the least significant bits of the data type (bits 0 through 7) and byte 7 being contained in the most significant bits (bits 56 through 63). The words in the packed words data type are numbered 0 through 4, with word 0 being contained in the bits 0 through 15 of the data type and word 4 being contained in bits 48 through 63. The doublewords in a packed doublewords data type are numbered 0 and 1, with doubleword 0 being contained in bits 0 through 31 and doubleword 1 being contained in bits 32 through 63.

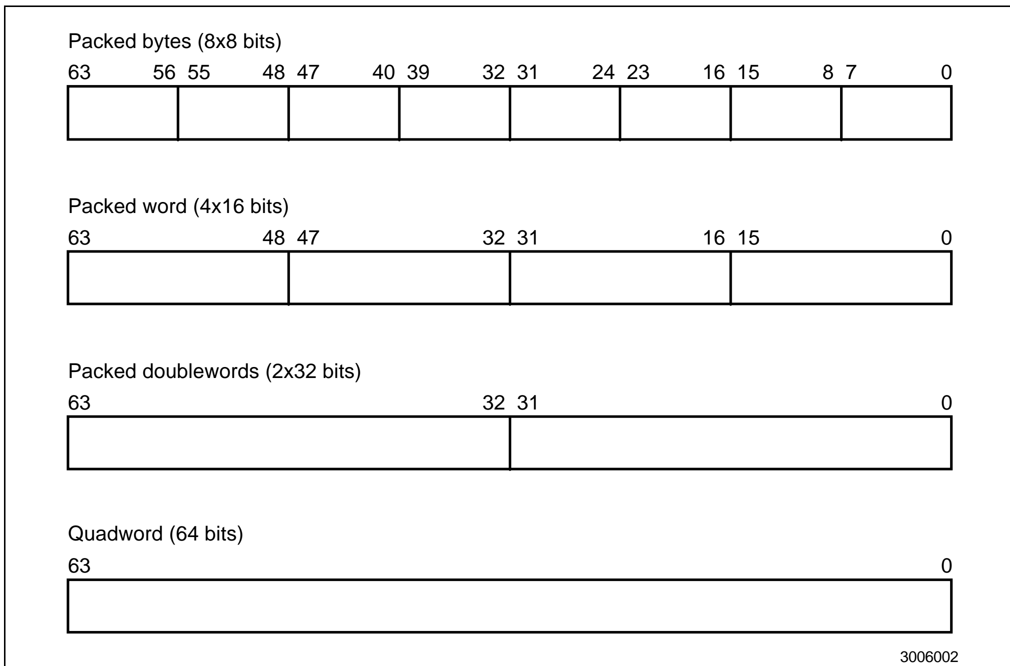


Figure 8-2. MMX™ Data Types

The MMX instructions move the packed data types (packed bytes, packed words, or packed doublewords) and the quadword data type to-and-from memory or to-and-from the Intel Architecture general-purpose registers in 64-bit blocks. However, when performing arithmetic or logical operations on the packed data types, the MMX instructions operate in parallel on the individual bytes, words, or doublewords contained in a 64-bit MMX register, as described in the following section (Section 8.1.3., "Single Instruction, Multiple Data (SIMD) Execution Model").

When operating on the bytes, words, and doublewords within packed data types, the MMX instructions recognize and operate on both signed and unsigned byte integers, word integers, and doubleword integers.

8.1.3. Single Instruction, Multiple Data (SIMD) Execution Model

The MMX technology uses the single instruction, multiple data (SIMD) technique for performing arithmetic and logical operations on the bytes, words, or doublewords packed into 64-bit MMX registers. For example, the PADDSB instruction adds 8 signed bytes from the source operand to 8 signed bytes in the destination operand and stores 8 byte-results in the destination operand. This SIMD technique speeds up software performance by allowing the same operation to be carried out on multiple data elements in parallel. The MMX technology supports parallel operations on byte, word, and doubleword data elements when contained in MMX registers.

The SIMD execution model supported in the MMX technology directly addresses the needs of modern media, communications, and graphics applications, which often use sophisticated algorithms that perform the same operations on a large number of small data types (bytes, words, and doublewords). For example, most audio data is represented in 16-bit (word) quantities. The MMX instructions can operate on 4 of these words simultaneously with one instruction. Video and graphics information is commonly represented as palletized 8-bit (byte) quantities. Here, one MMX instruction can operate on 8 of these bytes simultaneously.

8.1.4. Memory Data Formats

When stored in memory the bytes, words, and doublewords in the packed data types are stored in consecutive addresses, with the least significant byte, word, or doubleword being stored in the at the lowest address and the more significant bytes, words, or doubleword being stored at consecutively higher addresses (see Figure 8-3). The ordering bytes, words, or doublewords in memory is always little endian. That is, the bytes with the lower addresses are less significant than the bytes with the higher addresses.

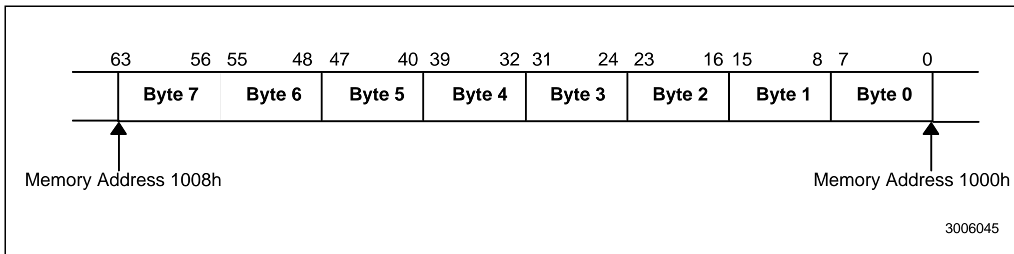


Figure 8-3. Eight Packed Bytes in Memory (at address 1000H)

8.1.5. Data Formats for MMX™ Registers

Values in MMX registers have the same format as a 64-bit quantity in memory. MMX registers have two data access modes: 64-bit access mode and 32-bit access mode.

The 64-bit access mode is used for 64-bit memory access, 64-bit transfer between MMX registers, all pack, logical and arithmetic instructions, and some unpack instructions.

The 32-bit access mode is used for 32-bit memory access, 32-bit transfer between integer registers and MMX registers, and some unpack instructions.

8.2. MMX™ INSTRUCTION SET

The MMX instruction set consists of 57 instructions, grouped into the following categories:

- Data transfer instructions
- Arithmetic instructions

- Comparison instructions
- Conversion instructions
- Logical instructions
- Shift instructions
- Empty MMX™ state instruction (EMMS)

When operating on packed data within an MMX register, the data is cast by the type specified by the instruction. For example, the PADDB (add packed bytes) instruction treats the packed data in an MMX register as 8 packed bytes; whereas, the PADDW (add packed words) instruction treats the packed data as 4 packed words.

8.2.1. Saturation Arithmetic and Wraparound Mode

The MMX technology supports a new arithmetic capability known as saturating arithmetic. Saturation is best defined by contrasting it with wraparound mode.

In wraparound mode, results that overflow or underflow are truncated and only the lower (least significant) bits of the result are returned; that is, the carry is ignored.

In saturation mode, results of an operation that overflow or underflow are clipped (saturated) to a data-range limit for the data type (see Table 8-1). The result of an operation that exceeds the range of a data-type saturates to the maximum value of the range. A result that is less than the range of a data type saturates to the minimum value of the range. This method of handling overflow and underflow is useful in many applications, such as color calculations.

Table 8-1. Data Range Limits for Saturation

Data Type	Lower Limit		Upper Limit	
	Hexadecimal	Decimal	Hexadecimal	Decimal
Signed Byte	80H	-128	7FH	127
Signed Word	8000H	-32,768	7FFFH	32,767
Unsigned Byte	00H	0	FFH	255
Unsigned Word	0000H	0	FFFFH	65,535

For example, when the result exceeds the data range limit for signed bytes, it is saturated to 7FH (FFH for unsigned bytes). If a value is less than the data range limit, it is saturated to 80H for signed bytes (00H for unsigned bytes).

Saturation provides a useful feature of avoiding wraparound artifacts. In the example of color calculations, saturation causes a color to remain pure black or pure white without allowing for and inversion.

MMX instructions do not indicate overflow or underflow occurrence by generating exceptions or setting flags.

8.2.2. Instruction Operands

All MMX instructions, except the EMMS instruction, reference and operate on two operands: the source and destination operands. The first operand is the destination and the second operand is the source. The destination operand may also be a second source operand for the operation. The instruction overwrites the destination operand with the result.

For example, a two-operand instruction would be decoded as:

DEST (first operand) ← DEST (first operand) OPERATION SRC (second operand)

The source operand for all the MMX instructions (except the data transfer instructions), can reside either in memory or in an MMX register. The destination operand resides in an MMX register.

For data transfer instructions, the source and destination operands can also be an integer register (for the MOVD instruction) or memory location (for both the MOVD and MOVQ instructions).

8.3. OVERVIEW OF THE MMX™ INSTRUCTION SET

Table 8-2 shows the instructions in the MMX instruction set. The following sections give a brief overview of each group of instructions in the MMX instruction set and the instructions within each group.

8.3.1. Data Transfer Instructions

The MOVD (Move 32 Bits) instruction transfers 32 bits of packed data from memory to MMX registers and visa versa, or from integer registers to MMX registers and visa versa.

The MOVQ (Move 64 Bits) instruction transfers 64-bits of packed data from memory to MMX registers and vise versa, or transfers data between MMX registers.

Table 8-2. MMX™ Instruction Set Summary

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		
		Packed	**Full Quadword**	
Logical	And		PAND	
	And Not		PANDN	
	Or		POR	
	Exclusive OR		PXOR	
Shift	Shift Left Logical	PSLLW, PSLLD	PSLLQ	
	Shift Right Logical	PSRLW, PSRLD	PSRLQ	
	Shift Right Arithmetic	PSRAW, PSRAD		
		Doubleword Transfers	**Quadword Transfers**	
Data Transfer	Register to Register	MOVD	MOVQ	
	Load from Memory	MOVD	MOVQ	
	Store to Memory	MOVD	MOVQ	
Empty MMX™ State		EMMS		

8.3.2. Arithmetic Instructions

The arithmetic instructions perform addition, subtraction, multiplication, and multiply/add operations on packed data types.

8.3.2.1. PACKED ADDITION AND SUBTRACTION

The PADDSB, PADDSW, and PADDWD (packed add) and PSUBB, PSUBW, and PSUBD (packed subtract) instructions add or subtract the signed or unsigned data elements of the source operand to or from the destination operand in wrap- around mode. These instructions support packed byte, packed word, and packed doubleword data types.

The PADDSB and PADDSW (packed add with saturation) and PSUBSB and PSUBSW (packed subtract with saturation) instructions add or subtract the signed data elements of the source operand to or from the signed data elements of the destination operand and saturate the result to the limits of the signed data-type range. These instructions support packed byte and packed word data types.

The PADDUSB and PADDUSW (packed add unsigned with saturation) and PSUBUSB and PSUBUSW (packed subtract unsigned with saturation) instructions add or subtract the unsigned data elements of the source operand to or from the unsigned data elements of the destination operand and saturate the result to the limits of the unsigned data-type range. These instructions support packed byte and packed word data types.

8.3.2.2. PACKED MULTIPLICATION

Packed multiplication instructions perform four multiplications on pairs of signed 16-bit operands, producing 32-bit intermediate results. Users may choose the low-order or high-order parts of each 32-bit result.

The PMULHW (packed multiply high) and PMULLW (packed multiply low) instructions multiply the signed words of the source and destination operands and write the high-order or low-order 16 bits of each of the results to the destination operand.

8.3.2.3. PACKED MULTIPLY ADD

The PMADDWD (packed multiply and add) instruction calculates the products of the signed words of the source and destination operands. The four intermediate 32-bit doubleword products are summed in pairs to produce two 32-bit doubleword results.

8.3.3. Comparison Instructions

The PCMPEQB, PCMPEQW, and PCMPEQD (packed compare for equal) and PCMPGTB, PCMPGTW, and PCMPGTD (packed compare for greater than) instructions compare the corresponding data elements in the source and destination operands for equality or value greater than, respectively. These instructions generate a mask of ones or zeros which are written to the destination operand. Logical operations can use the mask to select elements. This can be used to

implement a packed conditional move operation without a branch or a set of branch instructions. No flags are set.

These instructions support packed byte, packed word and packed doubleword data types.

8.3.4. Conversion Instructions

The conversion instructions convert the data elements within a packed data type.

The PACKSSWB and PACKSSDW (packed with signed saturation) instruction converts signed words into signed bytes or signed doublewords into signed words, in signed saturation mode.

The PACKUSWB (packed with unsigned saturation) instruction converts signed words into unsigned bytes, in unsigned saturation mode.

The PUNPCKHBW, PUNPCKHWD, and PUNPCKHDQ (unpack high packed data) and PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ (unpack low packed data) instructions convert bytes to words, words to doublewords, or doublewords to quadwords.

8.3.5. Logical Instructions

The PAND (bitwise logical AND), PANDN (bitwise logical AND NOT), POR (bitwise logical OR), and PXOR (bitwise logical exclusive OR) instructions perform bitwise logical operations on 64-bit quantities.

8.3.6. Shift Instructions

The logical shift left, logical shift right and arithmetic shift right instructions shift each element by a specified number of bits. The logical left and right shifts also enable a 64-bit quantity (quadword) to be shifted as one block, assisting in data type conversions and alignment operations.

The PSLLW and PSLLD (packed shift left logical) and PSRLW and PSRLD (packed shift right logical) instructions perform a logical left or right shift, and fill the empty high or low order bit positions with zeros. These instructions support packed word, packed doubleword, and quadword data types.

The PSRAW and PSRAD (packed shift right arithmetic) instruction performs an arithmetic right shift, copying the sign bit into empty bit positions on the upper end of the operand. This instruction supports packed word and packed doubleword data types.

8.3.7. EMMS (Empty MMX™ State) Instruction

The EMMS instruction empties the MMX state. This instruction must be used to clear the MMX state (empty the floating-point tag word) at the end of an MMX routine before calling other routines that can execute floating-point instructions.

8.4. COMPATIBILITY WITH FPU ARCHITECTURE

The MMX state is aliased upon the Intel Architecture floating-point state. No new state or mode is added to support the MMX technology. The same floating-point instructions that save and restore the floating-point state also handle the MMX state (for example, during context switching).

MMX technology uses the same interface techniques between the floating-point architecture and the operating system (primarily for task switching purposes). For more details, see Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*.

8.4.1. MMX™ Instructions and the Floating-Point Tag Word

After each MMX instruction, the entire floating-point tag word is set to Valid (00s). The Empty MMX state (EMMS) instruction sets the entire floating-point tag word to Empty (11s).

Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*, describes the effects of floating-point and MMX instructions on the floating-point tag word. For details on floating-point tag word, see Section 7.3.6., "FPU Tag Word".

8.4.2. Effect of Instruction Prefixes on MMX™ Instructions

Table 8-3 details the effect of an instruction prefix on an MMX instruction.

Table 8-3. Effect of Prefixes on MMX™ Instructions

Prefix Type	Effect of Prefix
Address size (67H)	Affects MMX™ instructions with a memory operand. Ignored by MMX instructions without a memory operand.
Operand size (66H)	Ignored.
Segment override	Affects MMX instructions with a memory operand. Ignored by MMX instructions without a memory operand.
Repeat	Ignored.
Lock (F0H)	Generates an invalid opcode exception.

See the section titled "Instruction Prefixes" in Chapter 2 of the *Intel Architecture Software Developer's Manual, Volume 2*, for detailed information on prefixes.

8.5. WRITING APPLICATIONS WITH MMX™ CODE

The following sections give guidelines for writing applications code uses the MMX technology.

8.5.1. Detecting Support for MMX™ Technology Using the CPUID Instruction

Use the CPUID instruction to determine whether the processor supports the MMX instruction set (see the section titled "CPUID—CPU Identification" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*, for a detailed description of the CPUID instruction). When the support for MMX technology is detected by the CPUID instruction, it is signaled by setting bit 23 (MMX technology bit) in the feature flags to 1. In general, two versions of the routine can be created: one with scalar instructions and one with MMX instructions. The application will call the appropriate routine depending on the results of the CPUID instruction. If support for MMX technology is detected, then the MMX routine is called; if no support for the MMX technology exists, the application calls the scalar routine.

NOTE

The CPUID instruction will continue to report the existence of the MMX technology if the CR0.EM bit is set (which signifies that the CPU is configured to generate exception interrupt 7 that can be used to emulate floating point instructions). In this case, executing an MMX instruction results in an invalid opcode exception.

Example 8-1 illustrates how to use the CPUID instruction. This example does not represent the entire CPUID sequence, but shows the portion used for detection of MMX technology.

Example 8-1. Partial Routine for Detecting MMX™ Technology with the CPUID Instruction

```
...                     ; identify existence of CPUID instruction
...
...                     ; identify Intel processor
....
mov   EAX, 1            ; request for feature flags
CPUID                   ; 0Fh, 0A2h CPUID instruction
test  EDX, 00800000h    ; Is IA MMX technology bit (Bit 23 of EDX)
                        ; in feature flags set?
jnz   MMX_Technology_Found
```

8.5.2. Using the EMMS Instruction

When integrating an MMX routine into an application running under an existing operating system, programmers need to take special precautions, similar to those when writing floating-point code.

When an MMX instruction executes, the floating-point tag word is marked valid (00s). Subsequent floating-point instructions that will be executed may produce unexpected results because the floating-point stack seems to contain valid data. The EMMS instruction marks the floating-

point tag word as empty. Therefore, it is imperative to use the EMMS instruction at the end of every MMX routine, if the next routine may contain FPU code.

The EMMS instruction must be used in each of the following cases:

- When an application using the floating-point instructions calls an MMX™ technology library/DLL. (Use the EMMS instruction at the end of the MMX code.)
- When an application using MMX instructions calls a floating-point library/DLL. (Use the EMMS instruction before calling the floating-point code.)
- When a switch is made between MMX code in a task/thread and other tasks/threads in cooperative operating systems, unless it is certain that more MMX instructions will be executed before any FPU code.

If the EMMS instruction is not used when trying to execute a floating-point instruction, the following may occur:

- Depending on the exception mask bits of the floating-point control word, a floating- point exception event may be generated.
- A "soft exception" may occur. In this case floating-point code continues to execute, but generates incorrect results. This happens when the floating-point exceptions are masked and no visible exceptions occur. The internal exception handler (microcode, not user visible) loads a NaN (Not a Number) with an exponent of 11..11B onto the floating-point stack. The NaN is used for further calculations, yielding incorrect results.
- A potential error may occur only if the operating system does NOT manage floating-point context across task switches. These operating systems are usually cooperative operating systems. It is imperative that the EMMS instruction execute at the end of all the MMX™ routines that may enable a task switch immediately after they end execution (explicit yield API or implicit yield API).

8.5.3. Interfacing with MMX™ Code

The MMX technology enables direct access to all the MMX registers. This means that all existing interface conventions that apply to the use of the processor's general-purpose registers (EAX, EBX, etc.) also apply to use of MMX register.

An efficient interface to MMX routines might pass parameters and return values through the MMX registers or through a combination of memory locations (via the stack) and MMX registers. Such an interface would have to be written in assembly language since passing parameters through MMX registers is not currently supported by any existing C compilers. Do not use the EMMS instruction when the interface to the MMX code has been defined to retain values in the MMX register.

If a high-level language, such as C, is used, the data types could be defined as a 64-bit structure with packed data types.

When implementing usage of MMX instructions in high-level languages other approaches can be taken, such as:

- Passing parameters to an MMX™ routine by passing a pointer to a structure via the integer stack.
- Returning a value from a function by returning the pointer to a structure.

8.5.4. Writing Code with MMX™ and Floating-Point Instructions

The MMX technology aliases the MMX registers on the floating-point registers. The main reason for this is to enable MMX technology to be fully compatible and transparent to existing software environments (operating systems and applications). This way operating systems will be able to include new applications and drivers that use the MMX technology.

An application can contain both floating-point and MMX code. However, the user is discouraged from causing frequent transitions between MMX and floating-point instructions by mixing MMX code and floating-point code.

8.5.4.1. RECOMMENDATIONS AND GUIDELINES

Do not mix MMX code and floating-point code at the instruction level for the following reasons:

- The TOS (top of stack) value of the floating-point status word is set to 0 after each MMX™ instruction. This means that the floating-point code loses its pointer to its floating-point registers if the code mixes MMX instructions within a floating-point routine.
- An MMX instruction write to an MMX register writes ones (11s) to the exponent part of the corresponding floating-point register.
- Floating-point code that uses register contents that were generated by the MMX instructions may cause floating-point exceptions or incorrect results. These floating-point exceptions are related to undefined floating-point values and floating-point stack usage.
- All MMX instructions (except EMMS) set the entire tag word to the valid state (00s in all tag fields) without preserving the previous floating-point state.
- Frequent transitions between the MMX and floating-point instructions may result in significant performance degradation in some implementations.

If the application contains floating-point and MMX instructions, follow these guidelines:

- Partition the MMX™ technology module and the floating-point module into separate instruction streams (separate loops or subroutines) so that they contain only instructions of one type.
- Do not rely on register contents across transitions.
- When the MMX state is not required, empty the MMX state using the EMMS instruction.
- Exit the floating-point code section with an empty stack.

Example 8-2. Floating-point (FP) and MMX™ Code

```
FP_code:
        ..
        ..            (*leave the FPU stack empty*)
MMX_code:
        ..
        EMMS          (*mark the FPU tag word as empty*)

FP_code 1:
        ..
        ..            (*leave the FPU stack empty*)
```

8.5.5. Using MMX™ Code in a Multitasking Operating System Environment

An application needs to identify the nature of the multitasking operating system on which it runs. Each task retains its own state which must be saved when a task switch occurs. The processor state (context) consists of the general-purpose registers and the floating-point and MMX registers.

Operating systems can be classified into two types:

- Cooperative multitasking operating system.
- Preemptive multitasking operating system.

The behavior of the two operating-system types in context switching is described in "Context Switching" in Chapter 10 of the *Intel Architecture Software Developer's Manual, Volume 3*.

8.5.5.1. COOPERATIVE MULTITASKING OPERATING SYSTEM

Cooperative multitasking operating systems do not save the FPU or MMX state when performing a context switch. Therefore, the application needs to save the relevant state before relinquishing direct or indirect control to the operating system.

8.5.5.2. PREEMPTIVE MULTITASKING OPERATING SYSTEM

Preemptive multitasking operating systems are responsible for saving and restoring the FPU and MMX state when performing a context switch. Therefore, the application does not have to save or restore the FPU and MMX state.

8.5.6. Exception Handling in MMX™ Code

MMX instructions generate the same type of memory-access exceptions as other Intel Architecture instructions. Some examples are: page fault, segment not present, and limit violations. Existing exception handlers can handle these types of exceptions. They do not have to be modified.

Unless there is a pending floating-point exception, MMX instructions do not generate numeric exceptions. Therefore, there is no need to modify existing exception handlers or add new ones.

If a floating-point exception is pending, the subsequent MMX instruction generates a numeric error exception (interrupt 16 and/or FERR#). The MMX instruction resumes execution upon return from the exception handler.

8.5.7. Register Mapping

The MMX registers and their tags are mapped to physical locations of the floating-point registers and their tags. Register aliasing and mapping is described in more detail in Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*.

intel®

9

Input/Output

CHAPTER 9
INPUT/OUTPUT

In addition to transferring data to and from external memory, Intel Architecture processors can also transfer data to and from input/output ports (I/O ports). I/O ports are created in system hardware by circuity that decodes the control, data, and address pins on the processor. These I/O ports are then configured to communicate with peripheral devices. An I/O port can be an input port, an output port, or a bidirectional port. Some I/O ports are used for transmitting data, such as to and from the transmit and receive registers, respectively, of a serial interface device. Other I/O ports are used to control peripheral devices, such as the control registers of a disk controller.

This chapter describes the processor's I/O architecture. The topics discussed include:

- I/O port addressing.
- I/O instructions.
- I/O protection mechanism.

9.1. I/O PORT ADDRESSING

The processor allows I/O ports to be accessed in either of two ways:

- Through a separate I/O address space.
- Through memory-mapped I/O.

Accessing I/O ports through the I/O address space is handled through a set of I/O instructions and a special I/O protection mechanism. Accessing I/O ports through memory-mapped I/O is handled with the processors general-purpose move and string instructions, with protection provided through segmentation or paging. I/O ports can be mapped so that they appear in the I/O address space or the physical-memory address space (memory mapped I/O) or both.

One benefit of using the I/O address space is that writes to I/O ports are guaranteed to be completed before the next instruction in the instruction stream is executed. Thus, I/O writes to control system hardware cause the hardware to be set to its new state before any other instructions are executed. See Section 9.6., "Ordering I/O", for more information on serializing of I/O operations.

9.2. I/O PORT HARDWARE

From a hardware point of view, I/O addressing is handled through the processor's address lines. For Pentium Pro processors, a special memory-I/O transaction on the system bus indicates whether the address lines are being driven with a memory address or an I/O address; for Pentium and earlier Intel Architecture processors, the M/IO pin indicates a memory address (1) or an I/O address (0). When the separate I/O address space is selected, it is the responsibility of the hardware to decode the memory-I/O bus transaction to select I/O ports rather than memory.

Data is transmitted between the processor and an I/O device through the data lines.

9.3. I/O ADDRESS SPACE

The processor's I/O address space is separate and distinct from the physical-memory address space. The I/O address space consists of 2^{16} (64K) individually addressable 8-bit I/O ports, numbered 0 through FFFFH. I/O port addresses 0F8H through 0FFH are reserved. Do not assign I/O ports to these addresses. The result of an attempt to address beyond the I/O address space limit of FFFFH is implementation-specific; see the Developer's Manuals for specific processors for more details.

Any two consecutive 8-bit ports can be treated as a 16-bit port, and any four consecutive ports can be a 32-bit port. In this manner, the processor can transfer 8, 16, or 32 bits to or from a device in the I/O address space. Like words in memory, 16-bit ports should be aligned to even addresses (0, 2, 4, ...) so that all 16 bits can be transferred in a single bus cycle. Likewise, 32-bit ports should be aligned to addresses that are multiples of four (0, 4, 8, ...). The processor supports data transfers to unaligned ports, but there is a performance penalty because one or more extra bus cycle must be used.

The exact order of bus cycles used to access unaligned ports is undefined and is not guaranteed to remain the same in future Intel Architecture processors. If hardware or software requires that I/O ports be written to in a particular order, that order must be specified explicitly. For example, to load a word-length I/O port at address 2H and then another word port at 4H, two word-length writes must be used, rather than a single doubleword write at 2H.

Note that the processor does not mask parity errors for bus cycles to the I/O address space. Accessing I/O ports through the I/O address space is thus a possible source of parity errors.

9.3.1. Memory-Mapped I/O

I/O devices that respond like memory components can be accessed through the processor's physical-memory address space (see Figure 9-1). When using memory-mapped I/O, any of the processor's instructions that reference memory can be used to access an I/O port located at a physical-memory address. For example, the MOV instruction can transfer data between any register and a memory-mapped I/O port. The AND, OR, and TEST instructions may be used to manipulate bits in the control and status registers of a memory-mapped peripheral devices.

When using memory-mapped I/O, caching of the address space mapped for I/O operations must be prevented. With the Pentium Pro processors, caching of I/O accesses can be prevented by using memory type range registers (MTRRs) to map the address space used for the memory-mapped I/O as uncacheable (UC). See Chapter 9, *Memory Cache Control*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for a complete discussion of the MTRRs.

The Pentium and Intel486 processors do not support MTRRs. Instead, they provide the KEN# pin, which when held inactive (high) prevents caching of all addresses sent out on the system bus. To use this pin, external address decoding logic is required to block caching in specific address spaces.

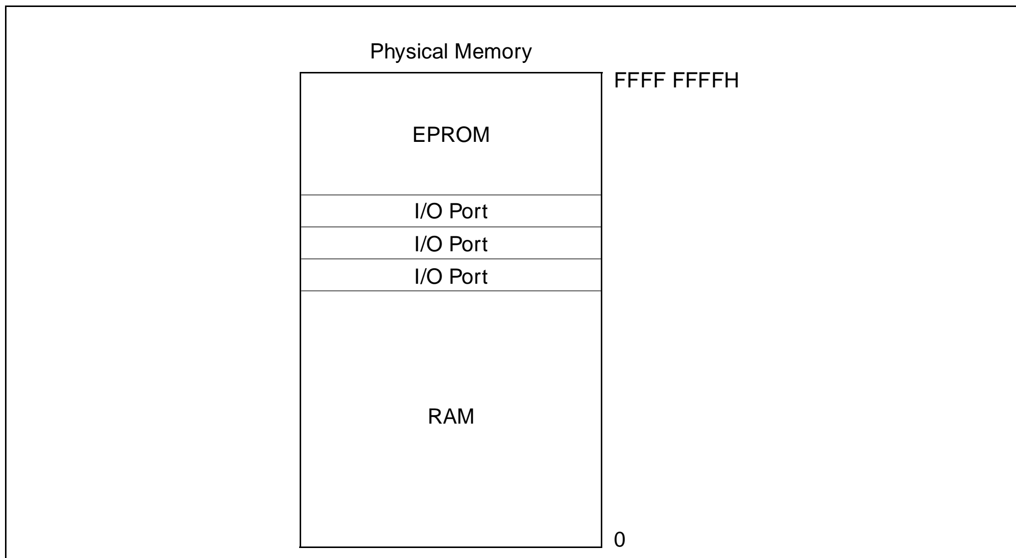


Figure 9-1. Memory-Mapped I/O

All the Intel Architecture processors that have on-chip caches also provide the PCD (page-level cache disable) flag in page table and page directory entries. This flag allows caching to be disabled on a page-by-page basis. See "Page-Directory and Page-Table Entries" in Chapter 3 of in the *Intel Architecture Software Developer's Manual, Volume 3*.

9.4. I/O INSTRUCTIONS

The processor's I/O instructions provide access to I/O ports through the I/O address space. (These instructions cannot be used to access memory-mapped I/O ports.) There are two groups of I/O instructions:

- Those which transfer a single item (byte, word, or doubleword) between an I/O port and a general-purpose register.
- Those which transfer strings of items (strings of bytes, words, or doublewords) between an I/O port and memory.

The register I/O instructions IN (input from I/O port) and OUT (output to I/O port) move data between I/O ports and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register. The address of the I/O port can be given with an immediate value or a value in the DX register.

The string I/O instructions INS (input string from I/O port) and OUTS (output string to I/O port) move data between an I/O port and a memory location. The address of the I/O port being accesses is given in the DX register; the source or destination memory address is given in the DS:ESI or ES:EDI register, respectively.

When used with one of the repeat prefixes (such as REP), the INS and OUTS instructions perform string (or block) input or output operations. The repeat prefix REP modifies the INS and OUTS instructions to transfer blocks of data between an I/O port and memory. Here, the ESI or EDI register is incremented or decremented (according to the setting of the DF flag in the EFLAGS register) after each byte, word, or doubleword is transferred between the selected I/O port and memory.

See the individual references for the IN, INS, OUT, and OUTS instructions in Chapter 3, *Instruction Set Reference*, of the *Intel Architecture Software Developer's Manual, Volume 2*, for more information on these instructions.

9.5. PROTECTED-MODE I/O

When the processor is running in protected mode, the following protection mechanisms regulate access to I/O ports:

- When accessing I/O ports through the I/O address space, two protection devices control access:
  - — The I/O privilege level (IOPL) field in the EFLAGS register.
  - — The I/O permission bit map of a task state segment (TSS).
- When accessing memory-mapped I/O ports, the normal segmentation and paging protection and the MTRRs (in processors that support them) also affect access to I/O ports. See Chapter 4, *Protection*, and Chapter 9, *Memory Cache Control*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for a complete discussion of memory protection.

The following sections describe the protection mechanisms available when accessing I/O ports in the I/O address space with the I/O instructions.

9.5.1. I/O Privilege Level

In systems where I/O protection is used, the IOPL field in the EFLAGS register controls access to the I/O address space by restricting use of selected instructions. This protection mechanism permits the operating system or executive to set the privilege level needed to perform I/O. In a typical protection ring model, access to the I/O address space is restricted to privilege levels 0 and 1. Here, kernel and the device drivers are allowed to perform I/O, while less privileged device drivers and application programs are denied access to the I/O address space. Application programs must then make calls to the operating system to perform I/O.

The following instructions can be executed only if the current privilege level (CPL) of the program or task currently executing is less than or equal to the IOPL: IN, INS, OUT, OUTS, CLI (clear interrupt-enable flag), and STI (set interrupt-enable flag). These instructions are called **I/O sensitive** instructions, because they are sensitive to the IOPL field. Any attempt by a less privileged program or task to use an I/O sensitive instruction results in a general-protection exception (#GP) being signaled. Because each task has its own copy of the EFLAGS register, each task can have a different IOPL.

The I/O permission bit map in the TSS can be used to modify the effect of the IOPL on I/O sensitive instructions, allowing access to some I/O ports by less privileged programs or tasks (see Section 9.5.2., "I/O Permission Bit Map").

A program or task can change its IOPL only with the POPF and IRET instructions; however, such changes are privileged. No procedure may change the current IOPL unless it is running at privilege level 0. An attempt by a less privileged procedure to change the IOPL does not result in an exception; the IOPL simply remains unchanged.

The POPF instruction also may be used to change the state of the IF flag (as can the CLI and STI instructions); however, the POPF instruction in this case is also I/O sensitive. A procedure may use the POPF instruction to change the setting of the IF flag only if the CPL is less than or equal to the current IOPL. An attempt by a less privileged procedure to change the IF flag does not result in an exception; the IF flag simply remains unchanged.

9.5.2. I/O Permission Bit Map

The I/O permission bit map is a device for permitting limited access to I/O ports by less privileged programs or tasks and for tasks operating in virtual-8086 mode. The I/O permission bit map is located in the TSS (see Figure 9-2) for the currently running task or program. The address of the first byte of the I/O permission bit map is given in the I/O map base address field of the TSS. The size of the I/O permission bit map and its location in the TSS are variable.

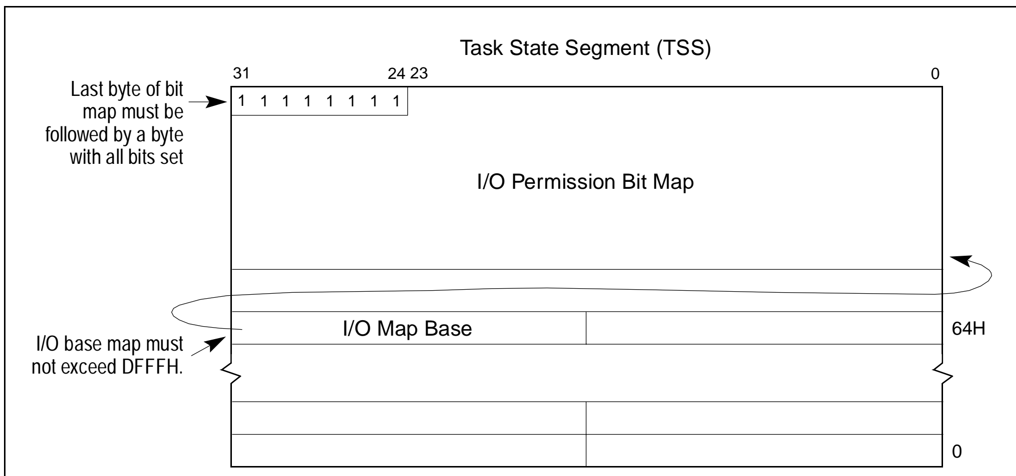


Figure 9-2. I/O Permission Bit Map

Because each task has its own TSS, each task has its own I/O permission bit map. Access to individual I/O ports can thus be granted to individual tasks.

If in protected mode and the CPL is less than or equal to the current IOPL, the processor allows all I/O operations to proceed. If the CPL is greater than the IOPL or if the processor is operating

in virtual-8086 mode, the processor checks the I/O permission bit map to determine if access to a particular I/O port is allowed. Each bit in the map corresponds to an I/O port byte address. For example, the control bit for I/O port address 29H in the I/O address space is found at bit position 1 of the sixth byte in the bit map. Before granting I/O access, the processor tests all the bits corresponding to the I/O port being addressed. For a doubleword access, for example, the processors tests the four bits corresponding to the four adjacent 8-bit port addresses. If any tested bit is set, a general-protection exception (#GP) is signaled. If all tested bits are clear, the I/O operation is allows to proceed.

Because I/O port addresses are not necessarily aligned to word and doubleword boundaries, the processor read two bytes from the I/O permission bit map for every access to an I/O port. To prevent exceptions from being generated when the ports with the highest addresses are accessed, an extra byte needs to included in the TSS immediately after the table. This byte must have all of its bits set, and it must be within the segment limit.

It is not necessary for the I/O permission bit map to represent all the I/O addresses. I/O addresses not spanned by the map are treated as if they had set bits in the map. For example, if the TSS segment limit is 10 bytes past the bit-map base address, the map has 11 bytes and the first 80 I/O ports are mapped. Higher addresses in the I/O address space generate exceptions.

If the I/O bit map base address is greater than or equal to the TSS segment limit, there is no I/O permission map, and all I/O instructions generate exceptions when the CPL is greater than the current IOPL. The I/O bit map base address must be less than or equal to DFFFH.

9.6. ORDERING I/O

When controlling I/O devices it is often important that memory and I/O operations be carried out in precisely the order programmed. For example, a program may write a command to an I/O port, then read the status of the I/O device from another I/O port. It is important that the status returned be the status of the device **after** it receives the command, not **before**.

When using memory-mapped I/O, caution should be taken to avoid situations in which the programmed order is not preserved by the processor. To optimize performance, the processor allows cacheable memory reads to be reordered ahead of buffered writes in most situations. Internally, processor reads (cache hits) can be reordered around buffered writes. When using memory-mapped I/O, therefore, is possible that an I/O read might be performed before the memory write of a previous instruction. The recommended method of enforcing program ordering of memory-mapped I/O accesses with the Pentium Pro processor is to use the MTRRs to make the memory mapped I/O address space uncacheable; for the Pentium and Intel486 processors, either the #KEN pin or the PCD flags can be used for this purpose (see Section 9.3.1., "Memory-Mapped I/O"). When the target of a read or write is in an uncacheable region of memory, memory reordering does not occur externally at the processor's pins (that is, reads and writes appear in-order). Designating a memory mapped I/O region of the address space as uncacheable insures that reads and writes of I/O devices are carried out in program order. See Chapter 9, *Memory Cache Control*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on using MTRRs.

Another method of enforcing program order is to insert one of the serializing instructions, such as the CPUID instruction, between operations. See Chapter 7, *Multiple Processor Management*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on serialization of instructions.

It should be noted that the chip set being used to support the processor (bus controller, memory controller, and/or I/O controller) may post writes to uncacheable memory which can lead to out-of-order execution of memory accesses. In situations where out-of-order processing of memory accesses by the chip set can potentially cause faulty memory-mapped I/O processing, code must be written to force synchronization and ordering of I/O operations. Serializing instructions can often be used for this purpose.

When the I/O address space is used instead of memory-mapped I/O, the situation is different in two respects:

- The processor never buffers I/O writes. Therefore, strict ordering of I/O operations is enforced by the processor. (As with memory-mapped I/O, it is possible for a chip set to post writes in certain I/O ranges.)
- The processor synchronizes I/O instruction execution with external bus activity (see Table 9-1).

Table 9-1. I/O Instruction Serialization

Instruction Being Executed	Processor Delays Execution of …		Until Completion of …	
	Current Instruction?	Next Instruction?	Pending Stores?	Current Store?
IN	Yes		Yes	
INS	Yes		Yes	
REP INS	Yes		Yes	
OUT		Yes	Yes	Yes
OUTS		Yes	Yes	Yes
REP OUTS		Yes	Yes	Yes

10

Processor Identification and Feature Determination

CHAPTER 10
PROCESSOR IDENTIFICATION AND FEATURE DETERMINATION

When writing software intended to run on several different types of Intel Architecture processors, it is generally necessary to identify the type of processor present in a system and the processor features that are available to an application. This chapter describes how to identify the processor that is executing the code and determine the features the processor supports. It also shows how to determine if an FPU or NPX is present. See Chapter 17, *Intel Architecture Compatibility*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for a complete list of the features that are available for the different Intel Architecture processors.

10.1. PROCESSOR IDENTIFICATION

The CPUID instruction returns the processor type for the processor that executes the instruction. It also indicates the features that are present in the processor, including the existence of an on-chip FPU. The following information can be obtained with this instruction:

- The highest operand value the instruction responds to (2 for the Pentium® Pro processors and 1 for the Pentium processors and recent Intel486™ processors).
- The processor's family identification (ID) number, model ID, and stepping ID.
- The presence of an on-chip FPU.
- Support for or the presence of the following architectural extensions and enhancements:
  - Virtual-8086 mode enhancements.
  - Debugging extensions.
  - Page-size extensions.
  - Read time stamp counter (RDTSC) instruction.
  - Read model specific registers (RDMSR) and write model specific registers (WRMSR) instructions.
  - Physical address extension.
  - Machine check exceptions.
  - Compare and exchange 8 bytes instruction (CMPXCHG8B).
  - On-chip, advanced programmable interrupt controller (APIC).
  - Memory-type range registers (MTRRs).
  - Page global flag.

— Machine check architecture.

— Conditional move instruction (CMOV*cc*).

— MMX™ technology.

- Cache and TLB information.

To use this instruction, a source operand value of 0, 1 or 2 is placed in the EAX register. Processor identification and feature information is then returned in the EAX, EBX, ECX, and EDX registers. See "CPUID—CPU Identification" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*, for more detailed information about the instruction.

AP-485*, Intel Processor Identification and the CPUID Instruction* (Order Number 241618), provides additional information and example source code for use in identifying Intel Architecture processors. It also contains guidelines for using the CPUID instruction to help maintain the widest range of software compatibility. The following guidelines are among the most important, and should always be followed when using the CPUID instruction to determine available features:

- Always begin by testing for the "GenuineIntel," message in the EBX, EDX, and ECX registers when the CPUID instruction is executed with EAX equal to 0. If the processor is not genuine Intel, the feature identification flags may have different meanings than are described in "CPUID—CPU Identification" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*.
- Do not assume a value of 1 in a feature identification flag indicates that a given feature is present. For future feature identification flags, a value of 1 may indicate that the specific feature is not present.
- Test feature identification flags individually and do not make assumptions about undefined bits.

Note that the CPUID instruction will cause the invalid opcode exception (#UD) if executed on a processor that does not support it. The CPUID instruction application note provides a code sequence to test the validity of the CPUID instruction. Also, this test code (for CPUID valid) is not reliable when executed in virtual-8086 mode. To avoid this, if the test code is written to run in real-address mode, the SMSW instruction must be used to read the PE bit from the MSW (lower half of CR0). If PE flag is set to 1, the Real Mode code is actually being executed in virtual-8086 mode, and the test sequence cannot be guaranteed to return reliable information. (Note that the new version of the CPUID application note (AP-485*, Intel Processor Identification and the CPUID Instruction* (Order Number 241618-005)), explains this virtual-8086 problem, but the older versions of the application note do not.)

10.2. IDENTIFICATION OF EARLIER INTEL ARCHITECTURE PROCESSORS

The CPUID instruction is only available in the Pentium Pro, Pentium, and recent Intel486 processors. For the earlier Intel Architecture processors (including the earlier Intel486 processors), several other architectural features can be exploited to identify the processor.

The settings of bits 12 and 13 (IOPL), 14 (NT), and 15 (reserved) in the EFLAGS register (see Figure 3-7) is different for Intel's 32-bit processors than for the Intel 8086 and Intel 286 processors. By examining the settings of these bits (with the PUSHF/PUSHFD and POP/POPFD instructions), an application program can determine whether the processor is an 8086, Intel286, or one of the Intel 32-bit processors:

- 8086 processor — Bits 12 through 15 of the EFLAGS register are always set.
- Intel 286 processor — Bits 12 through 15 are always clear in real-address mode.
- 32-bit processors — In real-address mode, bit 15 is always clear and bits 12 through 14 have the last value loaded into them. In protected mode, bit 15 is always clear, bit 14 has the last value loaded into it, and the IOPL bits depends on the current privilege level (CPL). The IOPL field can be changed only if the CPL is 0.

Other EFLAG register bits that can be used to differentiate between the 32-bit processors:

- Bit 18 (AC) — Implemented only on the Pentium® Pro, Pentium, and Intel486™ processors. The inability to set or clear this bit distinguishes an Intel386 processor from the other Intel 32-bit processors.
- Bit 21 (ID) — Determines if the processor is able to execute the CPUID instruction. The ability to set and clear this bit indicates that the processor is a Pentium Pro, Pentium, or later version Intel486 processor.

To determine whether an FPU or NPX is present in a system, applications can write to the FPU/NPX status and control registers using the FNINIT instruction and then verify the correct values are read back using the FNSTENV instruction.

After determining that an FPU or NPX is present, its type can then be determined. In most cases, the processor type will determine the type of FPU or NPX; however, an Intel386 processor is compatible with either an Intel 287 or Intel 387 math coprocessor. The method the coprocessor uses to represent ∞ (after the execution of the FINIT, FNINIT, or RESET instruction) indicates which coprocessor is present. The Intel 287 math coprocessor uses the same bit representation for $+\infty$ and $-\infty$; whereas, the Intel 387 math coprocessor uses different representations for $+\infty$ and $-\infty$.

EFLAGS Cross-Reference

APPENDIX A
EFLAGS CROSS-REFERENCE

The cross-reference in Table A-1 summarizes how the flags in the processor's EFLAGS register are affected by each instruction. For detailed information on how flags are affected, see Chapter 3, *Instruction Set Reference* in the *Intel Architecture Software Developer's Manual, Volume 2*. The following codes describe the how the flags are affected:

T	Instruction tests flag.
M	Instruction modifies flag (either sets or resets depending on operands).
0	Instruction resets flag.
1	Instruction sets flag.
—	Instruction's effect on flag is undefined.
R	Instruction restores prior value of flag.
Blank	Instruction does not affect flag.

Table A-1. EFLAGS Cross-Reference

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
AAA	—	—	—	TM	—	M					
AAD	—	M	M	—	M	—					
AAM	—	M	M	—	M	—					
AAS	—	—	—	TM	—	M					
ADC	M	M	M	M	M	TM					
ADD	M	M	M	M	M	M					
AND	0	M	M	—	M	0					
ARPL			M								
BOUND											
BSF/BSR	—	—	M	—	—	—					
BSWAP											
BT/BTS/BTR/BTC	—	—	—	—	—	M					

Table A-1. EFLAGS Cross-Reference (Contd.)

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
CALL											
CBW											
CLC						0					
CLD									0		
CLI								0			
CLTS											
CMC						M					
CMOV*cc*	T	T	T		T	T					
CMP	M	M	M	M	M	M					
CMPS	M	M	M	M	M	M			T		
CMPXCHG	M	M	M	M	M	M					
CMPXCHG8B			M								
CPUID											
CWD											
DAA	—	M	M	TM	M	TM					
DAS	—	M	M	TM	M	TM					
DEC	M	M	M	M	M						
DIV	—	—	—	—	—	—					
ENTER											
ESC											
FCMOV*cc*			T		T	T					
FCOMI, FCOMIP, FUCOMI, FUCOMIP			M		M	M					
HLT											
IDIV	—	—	—	—	—	—					
IMUL	M	—	—	—	—	M					
IN											
INC	M	M	M	M	M						
INS									T		
INT							0			0	
INTO	T						0			0	
INVD											
INVLPG											

Table A-1. EFLAGS Cross-Reference (Contd.)

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
IRET	R	R	R	R	R	R	R	R	R	T	
J*cc*	T	T	T		T	T					
JCXZ											
JMP											
LAHF											
LAR			M								
LDS/LES/LSS/LFS/LGS											
LEA											
LEAVE											
LGDT/LIDT/LLDT/LMSW											
LOCK											
LODS									T		
LOOP											
LOOPE/LOOPNE			T								
LSL			M								
LTR											
MOV											
MOV control, debug, test	—	—	—	—	—	—					
MOVS									T		
MOVSX/MOVZX											
MUL	M	—	—	—	—	M					
NEG	M	M	M	M	M	M					
NOP											
NOT											
OR	0	M	M	—	M	0					
OUT											
OUTS									T		
POP/POPA											
POPF	R	R	R	R	R	R	R	R	R	R	
PUSH/PUSHA/PUSHF											
RCL/RCR 1	M					TM					
RCL/RCR count	—					TM					

Table A-1. EFLAGS Cross-Reference (Contd.)

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
RDMSR											
RDPMC											
RDTSC											
REP/REPE/REPNE											
RET											
ROL/ROR 1	M					M					
ROL/ROR count	—					M					
RSM	M	M	M	M	M	M	M	M	M	M	M
SAHF		R	R	R	R	R					
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M					
SAL/SAR/SHL/SHR count	—	M	M	—	M	M					
SBB	M	M	M	M	M	TM					
SCAS	M	M	M	M	M	M			T		
SET*cc*	T	T	T		T	T					
SGDT/SIDT/SLDT/SMSW											
SHLD/SHRD	—	M	M	—	M	M					
STC						1					
STD									1		
STI								1			
STOS									T		
STR											
SUB	M	M	M	M	M	M					
TEST	0	M	M	—	M	0					
UD2											
VERR/VERRW			M								
WAIT											
WBINVD											
WRMSR											
XADD	M	M	M	M	M	M					
XCHG											
XLAT											
XOR	0	M	M	—	M	0					

B

EFLAGS Condition Codes

APPENDIX B
EFLAGS CONDITION CODES

Table B-1 gives all the condition codes that can be tested for by the CMOV*cc*, FCMOV*cc*, J*cc* and SET*cc* instructions. The condition codes refer to the setting of one or more status flags (CF, OF, SF, ZF, and PF) in the EFLAGS register. The "Mnemonic" column gives the suffix (*cc*) added to the instruction to specific the test condition. The "Condition Tested For" column describes the condition specified in the "Status Flags Setting" column. The "Instruction Subcode" column gives the opcode suffix added to the main opcode to specify a test condition.

Table B-1. EFLAGS Condition Codes

Mnemonic (*cc*)	Condition Tested For	Instruction Subcode	Status Flags Setting
O	Overflow	0000	OF = 1
NO	No overflow	0001	OF = 0
B NAE	Below Neither above nor equal	0010	CF = 1
NB AE	Not below Above or equal	0011	CF = 0
E Z	Equal Zero	0100	ZF = 1
NE NZ	Not equal Not zero	0101	ZF = 0
BE NA	Below or equal Not above	0110	(CF OR ZF) = 1
NBE A	Neither below nor equal Above	0111	(CF OR ZF) = 0
S	Sign	1000	SF = 1
NS	No sign	1001	SF = 0
P PE	Parity Parity even	1010	PF = 1
NP PO	No parity Parity odd	1011	PF = 0
Mnemonic	Meaning	Instruction Subcode	Condition Tested
L NGE	Less Neither greater nor equal	1100	(SF xOR OF) = 1
NL GE	Not less Greater or equal	1101	(SF xOR OF) = 0

Table B-1. EFLAGS Condition Codes (Contd.)

Mnemonic (*cc*)	Condition Tested For	Instruction Subcode	Status Flags Setting
LE NG	Less or equal Not greater	1110	((SF XOR OF) OR ZF) = 1
NLE G	Neither less nor equal Greater	1111	((SF XOR OF) OR ZF) = 0

Many of the test conditions are described in two different ways. For example LE (less or equal) and NG (not greater) describe the same test condition. Alternate mnemonics are provided to make code more intelligible.

The terms “above” and “below” are associated with the CF flag and refer to the relation between two unsigned integer values. The terms “greater” and “less” are associated with the SF and OF flags and refer to the relation between two signed integer values.

C

Floating-Point Exceptions Summary

APPENDIX C
FLOATING-POINT EXCEPTIONS SUMMARY

Table C-1 lists the floating-point instruction mnemonics in alphabetical order. For each mnemonic, it summarizes the exceptions that the instruction may cause. See Section 7.8., "Floating-Point Exception Conditions", for a detailed discussion of the floating-point exceptions. The following codes indicate the floating-point exceptions:

#IS	Invalid-operation exception for stack underflow or stack overflow.
#IA	Invalid-operation exception for invalid arithmetic operands and unsupported formats.
#D	Denormal-operand exception.
#Z	Divide-by-zero exception.
#O	Numeric-overflow exception.
#U	Numeric-underflow exception.
#P	Inexact-result (precision) exception.

Table C-1. Floating-Point Exceptions Summary

Mnemonic	Instruction	#IS	#IA	#D	#Z	#O	#U	#P
F2XM1	2^X-1	Y	Y	Y			Y	Y
FABS	Absolute value	Y						
FADD(P)	Add real	Y	Y	Y		Y	Y	Y
FBLD	BCD load	Y						
FBSTP	BCD store and pop	Y	Y					Y
FCHS	Change sign	Y						
FCLEX	Clear exceptions							
FCMOV*cc*	Floating-point conditional move	Y						
FCOM, FCOMP, FCOMPP	Compare real	Y	Y	Y				
FCOMI, FCOMIP, FUCOMI, FUCOMIP	Compare real and set EFLAGS	Y	Y					
FCOS	Cosine	Y	Y	Y			Y	Y
FDECSTP	Decrement stack pointer							
FDIV(R)(P)	Divide real	Y	Y	Y	Y	Y	Y	Y
FFREE	Free register							

Table C-1. Floating-Point Exceptions Summary (Contd.)

Mnemonic	Instruction	#IS	#IA	#D	#Z	#O	#U	#P
FIADD	Integer add	Y	Y	Y		Y	Y	Y
FICOM(P)	Integer compare	Y	Y	Y				
FIDIV	Integer divide	Y	Y	Y	Y		Y	Y
FIDIVR	Integer divide reversed	Y	Y	Y	Y	Y	Y	Y
FILD	Integer load	Y						
FIMUL	Integer multiply	Y	Y	Y		Y	Y	Y
FINCSTP	Increment stack pointer							
FINIT	Initialize processor							
FIST(P)	Integer store	Y	Y					Y
FISUB(R)	Integer subtract	Y	Y	Y		Y	Y	Y
FLD extended or stack	Load real	Y						
FLD single or double	Load real	Y	Y	Y				
FLD1	Load + 1.0	Y						
FLDCW	Load Control word	Y	Y	Y	Y	Y	Y	Y
FLDENV	Load environment	Y	Y	Y	Y	Y	Y	Y
FLDL2E	Load $\log_2 e$	Y						
FLDL2T	Load $\log_2 10$	Y						
FLDLG2	Load $\log_{10} 2$	Y						
FLDLN2	Load $\log_e 2$	Y						
FLDPI	Load π	Y						
FLDZ	Load + 0.0	Y						
FMUL(P)	Multiply real	Y	Y	Y		Y	Y	Y
FNOP	No operation							
FPATAN	Partial arctangent	Y	Y	Y			Y	Y
FPREM	Partial remainder	Y	Y	Y			Y	
FPREM1	IEEE partial remainder	Y	Y	Y			Y	
FPTAN	Partial tangent	Y	Y	Y			Y	Y
FRNDINT	Round to integer	Y	Y	Y				Y
FRSTOR	Restore state	Y	Y	Y	Y	Y	Y	Y
FSAVE	Save state							
FSCALE	Scale	Y	Y	Y		Y	Y	Y
FSIN	Sine	Y	Y	Y			Y	Y
FSINCOS	Sine and cosine	Y	Y	Y			Y	Y

Table C-1. Floating-Point Exceptions Summary (Contd.)

Mnemonic	Instruction	#IS	#IA	#D	#Z	#O	#U	#P
FSQRT	Square root	Y	Y	Y				Y
FST(P) stack or extended	Store real	Y						
FST(P) single or double	Store real	Y	Y	Y		Y	Y	Y
FSTCW	Store control word							
FSTENV	Store environment							
FSTSW (AX)	Store status word							
FSUB(R)(P)	Subtract real	Y	Y	Y		Y	Y	Y
FTST	Test	Y	Y	Y				
FUCOM(P)(P)	Unordered compare real	Y	Y	Y				
FWAIT	CPU Wait							
FXAM	Examine							
FXCH	Exchange registers	Y						
FXTRACT	Extract	Y	Y	Y	Y			
FYL2X	$Y \cdot \log_2 X$	Y	Y	Y	Y	Y	Y	Y
FYL2XP1	$Y \cdot \log_2(X + 1)$	Y	Y	Y			Y	Y

D

Guidelines for Writing FPU Exception Handlers

APPENDIX D GUIDELINES FOR WRITING FPU EXCEPTION HANDLERS

As described in Chapter 7, *Floating-Point Unit*, the Intel Architecture supports two mechanisms for accessing exception handlers to handle unmasked FPU exceptions: native mode and MS-DOS compatibility mode. The primary purpose of this appendix is to provide detailed information to help software engineers design and write FPU exception-handling facilities to run on PC systems that use the MS-DOS compatibility mode[1] for handling FPU exceptions. Some of the information in this appendix will also be of interest to engineers who are writing native-mode FPU exception handlers. The information provided is as follows:

- Discussion of the origin of the MS-DOS* FPU exception handling mechanism and its relationship to the FPU's native exception handling mechanism.
- Description of the Intel Architecture flags and processor pins that control the MS-DOS FPU exception handling mechanism.
- Description of the external hardware typically required to support MS-DOS exception handling mechanism.
- Description of the FPU's exception handling mechanism and the typical protocol for FPU exception handlers.
- Code examples that demonstrate various levels of FPU exception handlers.
- Discussion of FPU considerations in multitasking environments.
- Discussion of native mode FPU exception handling.

The information given is oriented toward the most recent generations of Intel architecture processors, starting with the Intel486. It is intended to augment the reference information given in Chapter 7, *Floating-Point Unit*.

A more extensive version of this appendix is available in the application note AP-578, *Software and Hardware Considerations for FPU Exception Handlers for Intel Architecture Processors* (Order Number 242415-001), which is available from Intel.

---

1. Microsoft Windows* 95 and Windows* 3.1 (and earlier versions) operating systems use almost the same FPU exception handling interface as the operating system. The recommendations in this appendix for a MS-DOS* compatible exception handler thus apply to all three operating systems.

D.1. ORIGIN OF THE MS-DOS* COMPATIBILITY MODE FOR HANDLING FPU EXCEPTIONS

The first generations of Intel Architecture processors (starting with the Intel 8086 and 8088 processors and going through the Intel 286 and Intel386 processors) did not have an on-chip floating-point unit. Instead, floating-point capability was provided on a separate numeric coprocessor chip. The first of these numeric coprocessors was the Intel 8087, which was followed by the Intel 287 and Intel 387 numeric coprocessors.

To allow the 8087 to signal floating-point exceptions to its companion 8086 or 8088, the 8087 has an output pin, INT, which it asserts when an unmasked floating-point exception occurs. The designers of the 8087 recommended that the output from this pin be routed through a programmable interrupt controller (PIC) such as the Intel 8259A to the INTR pin of the 8086 or 8088. The accompanying interrupt vector number could then be used to access the floating-point exception handler.

However, the original IBM PC design and MS-DOS operating system used a different mechanism for handling the INT output from the 8087. It connected the INT pin directly to the NMI input pin of the 8086 or 8088. The NMI interrupt handler then had to determine if the interrupt was caused by a floating-point exception or another NMI event. This mechanism is the origin of want is now called the “MS-DOS compatibility mode.” The decision to use this latter floating-point exception handling mechanism came about because when the IBM PC was first designed, the 8087 was not available. When the 8087 did become available, other functions had already been assigned to the eight inputs to the PIC. One of these functions was a BIOS video interrupt, which was assigned to interrupt number 16 for the 8086 and 8088.

The Intel 286 processor created the “native mode” for handling floating-point exceptions by providing a dedicated input pin (ERROR#) for receiving floating-point exception signals and a dedicated interrupt number, 16. Interrupt 16 was used to signal floating-point errors (also called math faults). It was intended that the ERROR# pin on the Intel 286 be connected to a corresponding ERROR# pin on the Intel 287 numeric coprocessor. When the Intel 287 signals a floating-point exception using this mechanism, the Intel 286 generates an interrupt 16, to invoke the floating-point exception handler.

To maintain compatibility existing PC software, the native floating-point exception handling mode of the Intel 286 and 287 was not used in the IBM PC AT* system design. Instead, the ERROR# pin on the Intel 286 was tied permanently high, and the ERROR# pin from the Intel 287 was routed to a second (cascaded) PIC. The resulting output of this PIC was routed through an exception handler and eventually caused an interrupt 2 (NMI interrupt). Here the NMI interrupt was shared with PC AT’s new parity checking feature. Interrupt 16 remained assigned to the BIOS video interrupt handler. The external hardware for the MS-DOS compatibility mode must prevent the Intel 286 processor from executing past the next FPU instruction when an unmasked exception has been generated. To do this, it asserts the BUSY# signal into the Intel 286 when the ERROR# signal is asserted by the Intel 287.

The Intel386 processor and its companion Intel 387 numeric coprocessor provided the same hardware mechanism for signaling and handling floating-point exceptions as the Intel 286 and 287 processors. And again, to maintain compatibility with existing MS-DOS software, basically the same MS-DOS compatibility floating-point exception handling mechanism that was used in the PC AT was used in PCs based on the Intel386.

D.2. IMPLEMENTATION OF THE MS-DOS* COMPATIBILITY MODE IN THE INTEL486™, PENTIUM®, AND PENTIUM PRO PROCESSORS

Beginning with the Intel486 processor, the Intel Architecture provided a dedicated mechanism for enabling the MS-DOS compatibility mode for FPU exceptions and for generating external FPU-exception signals while operating in this mode. The following sections describe the implementation of the MS-DOS compatibility mode in Intel486 and Pentium processors and in the Pentium Pro processor. Also described is the recommended external hardware to support this mode of operation.

D.2.1. MS-DOS* Compatibility Mode in the Intel486™ and Pentium® Processors

In the Intel486, several things were done to enhance and speed up the numeric coprocessor, now called the floating-point unit (FPU). The most important enhancement was that the FPU was included in the same chip as the processor, for increased speed in FPU computations and reduced latency for FPU exception handling. Also, for the first time, the MS-DOS compatibility mode was built into the chip design, with the addition of the NE bit in control register CR0 and the addition of the FERR# (Floating point ERRor) and IGNNE# (IGNore Numeric Error) pins.

The NE bit selects the native FPU exception handling mode (NE = 1) or the MS-DOS compatibility mode (NE = 0). When native mode is selected, all signaling of floating-point exceptions is handled internally in the Intel486 chip, resulting in the generation of an interrupt 16.

When MS-DOS compatibility mode is selected the FERRR# and IGNNE# pins are used to signal floating-point exceptions. The FERR# output pin, which replaces the ERROR# pin from the previous generations of Intel Architecture numeric coprocessors, is connected to a PIC. A new input signal, IGNNE#, is provided to allow the FPU exception handler to execute FPU instructions, if desired, without first clearing the error condition and without triggering the interrupt a second time. This IGNNE# feature is needed to replicate the capability that was provided on MS-DOS compatibility Intel 286 and Intel 287 and Intel386 and Intel 387 systems by turning off the BUSY# signal, when inside the FPU exception handler, before clearing the error condition.

Note that Intel, in order to provide Intel486 processors for market segments which had no need for an FPU, created the “SX” versions. These Intel486 SX processors did not contain the floating point unit. Intel also produced Intel 487 SX processors for end users who later decided to upgrade to a system with an FPU. These Intel 487 SX processors are similar to standard Intel486 processors with a working FPU on board. Thus, the external circuitry necessary to support the MS-DOS compatibility mode for Intel 487 SX processors is the same as for standard Intel486 DX processors.

The Pentium and Pentium Pro processors offer the same mechanism (the NE bit and the FERR# and IGNNE# pins) as the Intel486 processors for generating FPU exceptions in MS-DOS compatibility mode. The actions of these mechanisms are slightly different and more straight-

forward for the Pentium Pro processors, as described in Section D.2.2., “MS-DOS* Compatibility Mode in the Pentium® Pro Processor”.

For Pentium and Pentium Pro processors, it is important to note that the special DP (Dual Processing) mode for Pentium Processors and also the more general Intel MultiProcessor Specification for systems with multiple Pentium or Pentium Pro processors support FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatibility FPU mode for systems using more than one processor.

D.2.1.1. BASIC RULES: WHEN FERR# IS GENERATED

When MS-DOS compatibility mode is enabled for the Intel486 or Pentium processors (NE bit is set to 0) and the IGNNE# input pin is de-asserted, the FERR# signal is generated as follows:

1. When an FPU instruction causes an unmasked FPU exception, the processor (in most cases) uses a “deferred” method of reporting the error. This means that the processor does not respond immediately, but rather freezes just before executing the next WAIT or FPU instruction (except for “no-wait” instructions, which the FPU executes regardless of an error condition).
2. When the processor freezes, it also asserts the FERR# output.
3. The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# assertion.
4. In MS-DOS* compatibility systems, FERR# is fed to the IRQ13 input in the cascaded PIC. The PIC generates interrupt 75H, which then branches to interrupt 2, as described earlier in this appendix for systems using the Intel 286 and Intel 287 or Intel386 and Intel 387 processors.

The deferred method of error reporting is used for all exceptions caused by the basic arithmetic instructions (including FADD, FSUB, FMUL, FDIV, FSQRT, FCOM and FUCOM), for precision exceptions caused by all types of FPU instructions, and for numeric underflow and overflow exceptions caused by all types of FPU instructions except stores to memory.

Some FPU instructions with some FPU exceptions use an “immediate” method of reporting errors. Here, the FERR# is asserted immediately, at the time that the exception occurs. The immediate method of error reporting is used for FPU stack fault, invalid operation and denormal exceptions caused by all transcendental instructions, FSCALE, FXTRACT, FPREM and others, and all exceptions (except precision) when caused by FPU store instructions. Like deferred error reporting, immediate error reporting will cause the processor to freeze just before executing the next WAIT or FPU instruction if the error condition has not been cleared by that time.

Note that in general, whether deferred or immediate error reporting is used for an FPU exception depends both on which exception occurred and which instruction caused that exception. A complete specification of these cases, which applies to both the Pentium and the Intel486 processors, is given in Section 5.1.21 in the *Pentium® Processor Family Developer’s Manual: Volume 1*.

If NE=0 but the IGNNE# input is active while an unmasked FPU exception is in effect, the processor disregards the exception, does not assert FERR#, and continues. If IGNNE# is then de-asserted and the FPU exception has not been cleared, the processor will respond as described above. (That is, an immediate exception case will assert FERR# immediately. A deferred exception case will assert FERR# and freeze just before the next FPU or WAIT instruction.) The assertion of IGNNE# is intended for use only inside the FPU exception handler, where it is needed if one wants to execute non-control FPU instructions for diagnosis, before clearing the exception condition. When IGNNE# is asserted inside the exception handler, a preceding FPU exception has already caused FERR# to be asserted, and the external interrupt hardware has responded, but IGNNE# assertion still prevents the freeze at FPU instructions. Note that if IGNNE# is left active outside of the FPU exception handler, additional FPU instructions may be executed after a given instruction has caused an FPU exception. In this case, if the FPU exception handler ever did get invoked, it could not determine which instruction caused the exception.

To properly manage the interface between the processor's FERR# output, its IGNNE# input, and the IRQ13 input of the PIC, additional external hardware is needed. A recommended configuration is described in the following section.

D.2.1.2. RECOMMENDED EXTERNAL HARDWARE TO SUPPORT THE MS-DOS* COMPATIBILITY MODE

Figure D-1 provides an external circuit that will assure proper handling of FERR# and IGNNE# when an FPU exception occurs. In particular, it assures that IGNNE# will be active only inside the FPU exception handler without depending on the order of actions by the exception handler. Some hardware implementations have been less robust because they have depended on the exception handler to clear the FPU exception interrupt request to the PIC (FP_IRQ signal) **before** the handler causes FERR# to be de-asserted by clearing the exception from the FPU itself. Figure D-2 shows the details of how IGNNE# will behave when the circuit in Figure D-1 is implemented. The temporal regions within the FPU exception handler activity are described as follows:

1. The FERR# signal is activated by an FPU exception and sends an interrupt request through the PIC to the processor's INTR pin.

2. During the FPU interrupt service routine (exception handler) the processor will need to clear the interrupt request latch (Flip Flop #1). It may also want to execute non-control FPU instructions before the exception is cleared from the FPU. For this purpose the IGNNE# must be driven low. Typically in the PC environment an I/O access to Port 0F0H clears the external FPU exception interrupt request (FP_IRQ). In the recommended circuit, this access also is used to activate IGNNE#. With IGNNE# active the FPU exception handler may execute any FPU instruction without being blocked by an active FPU exception.

3. Clearing the exception within the FPU will cause the FERR# signal to be deactivated and then there is no further need for IGNNE# to be active. In the recommended circuit, the deactivation of FERR# is used to deactivate IGNNE#. If another circuit is used, the software and circuit together must assure that IGNNE# is deactivated no later than the exit from the FPU exception handler.

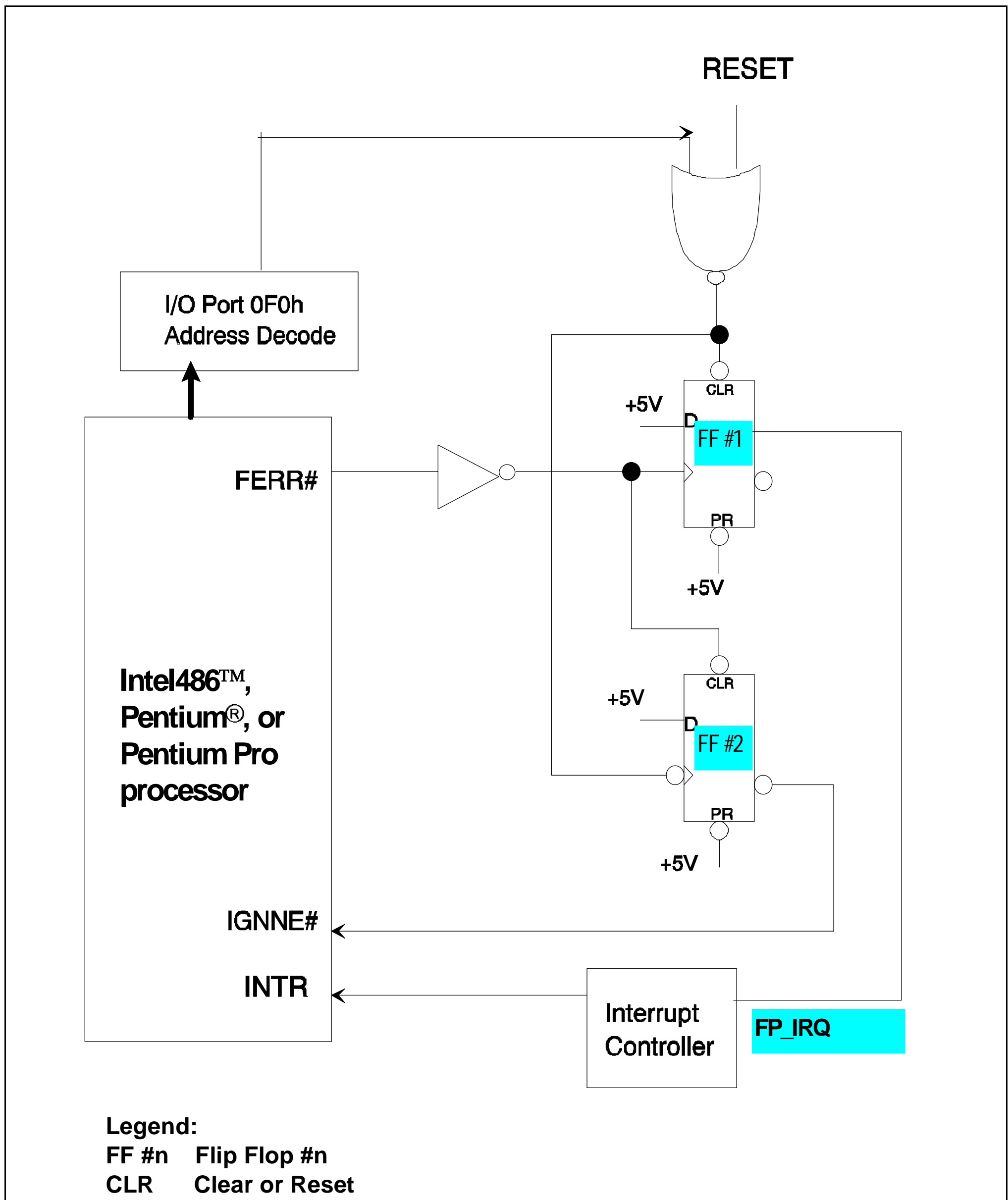


Figure D-1. Recommended Circuit for MS-DOS* Compatibility FPU Exception Handling

In the circuit in Figure D-1, when the FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. So the handler can activate IGNNE#, if needed, by doing this 0F0H access before clearing the FPU exception condition (which de-asserts FERR#). However, the circuit does not depend on the order of actions by the FPU exception handler to guarantee the correct hardware state upon exit from the handler. Flip Flop #2, which drives IGNNE# to the processor, has its CLEAR input attached to the inverted FERR#. This ensures that IGNNE# can never be active when FERR# is inactive. So if the handler clears the FPU exception condition **before** the 0F0H access, IGNNE# does not get activated and left on after exit from the handler.

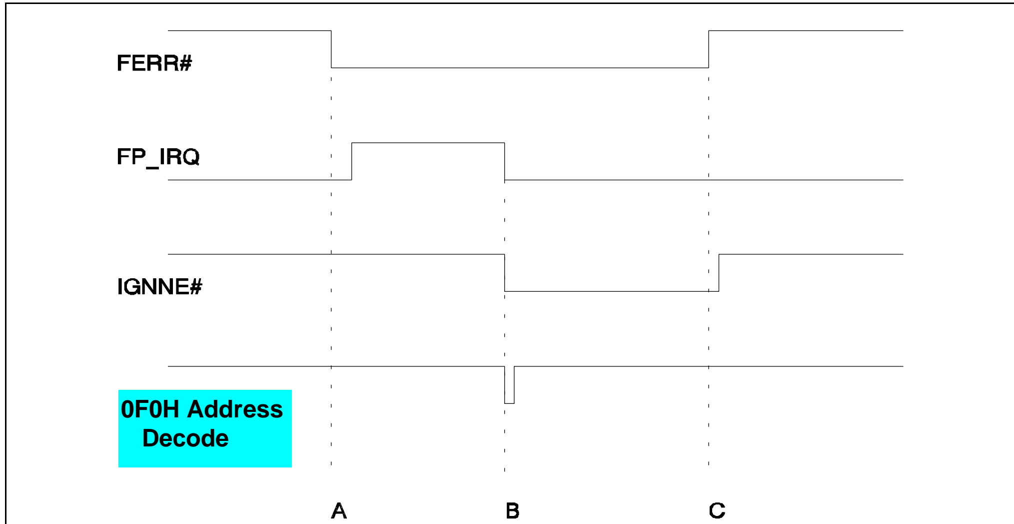


Figure D-2. Behavior of Signals During FPU Exception Handling

D.2.1.3. NO-WAIT FPU INSTRUCTIONS CAN GET FPU INTERRUPT IN WINDOW

The Pentium and Intel486 processors implement the “no-wait” floating-point instructions (FNINIT, FNCLEX, FNSTENV, FNSAVE, FNSTSW, FNSTCW, FNENI, FNDISI or FNSETPM) in the MS-DOS compatibility mode in the following manner. (See Section 7.5.11., “FPU Control Instructions” and Section 7.5.12., “Waiting Vs. Non-waiting Instructions” for a discussion of the no-wait instructions.)

If an unmasked numeric exception is pending from a preceding FPU instruction, a member of the no-wait class of instructions will, at the beginning of its execution, assert the FERR# pin in response to that exception just like other FPU instructions, but then, unlike the other FPU instructions, FERR# will be de-asserted. This de-assertion was implemented to allow the no-wait class of instructions to proceed without an interrupt due to any pending numeric exception. However, the brief assertion of FERR# is sufficient to latch the FPU exception request into most hardware interface implementations (including Intel’s recommended circuit).

All the FPU instructions are implemented such that during their execution, there is a window in which the processor will sample and accept external interrupts. If there is a pending interrupt, the processor services the interrupt first before resuming the execution of the instruction. Consequently, it is possible that the no-wait floating-point instruction may accept the external interrupt caused by it's own assertion of the FERR# pin in the event of a pending unmasked numeric exception, which is not an explicitly documented behavior of a no-wait instruction. This process is illustrated in Figure D-3.

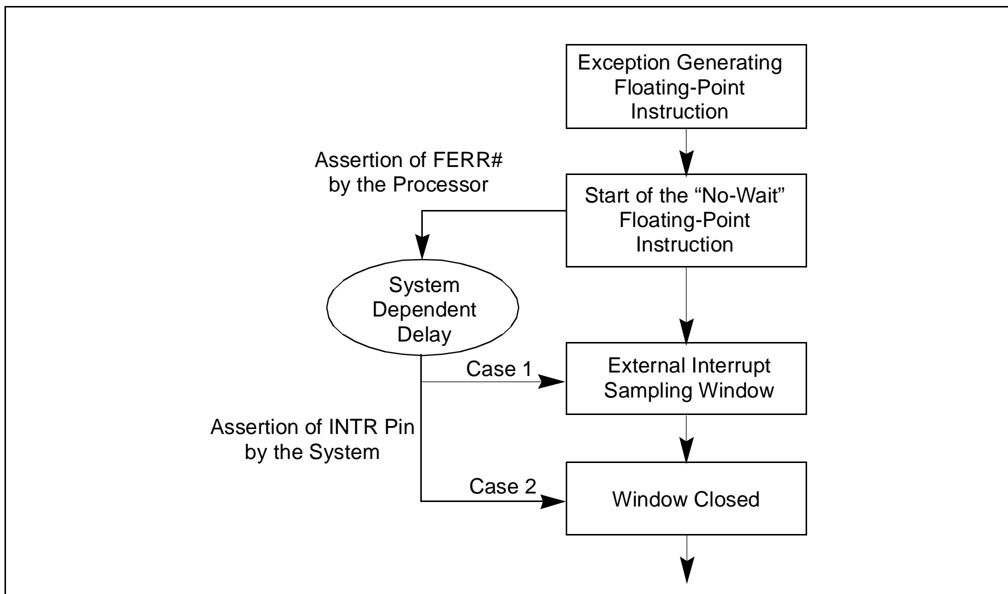


Figure D-3. Timing of Receipt of External Interrupt

Figure D-3 assumes that a floating-point instruction that generates a "deferred" error (as defined in the Section D.2.1.1., "Basic Rules: When FERR# Is Generated"), which asserts the FERR# pin only on encountering the next floating-point instruction, causes an unmasked numeric exception. Assume that the next floating-point instruction following this instruction is one of the no-wait floating-point instructions. The FERR# pin is asserted by the processor to indicate the pending exception on encountering the no-wait floating-point instruction. After the assertion of the FERR# pin the no-wait floating-point instruction opens a window where the pending external interrupts are sampled.

Then there are two cases possible depending on the timing of the receipt of the interrupt via the INTR pin (asserted by the system in response to the FERR# pin) by the processor.

Case 1 If the system responds to the assertion of FERR# pin by the no-wait floating-point instruction via the INTR pin during this window then the interrupt is serviced first, before resuming the execution of the no-wait floating-point instruction.

Case 2 If the system responds via the INTR pin after the window has closed then the interrupt is recognized only at the next instruction boundary.

There are two other ways, in addition to Case 1 above, in which a no-wait floating-point instruction can service a numeric exception inside its interrupt window. First, the first floating-point error condition could be of the "immediate" category (as defined in Section D.2.1.1., "Basic Rules: When FERR# Is Generated") that asserts FERR# immediately. If the system delay before asserting INTR is long enough, relative to the time elapsed before the no-wait floating-point instruction, INTR can be asserted inside the interrupt window for the latter. Second, consider two no-wait FPU instructions in close sequence, and assume that a previous FPU instruction has caused an unmasked numeric exception. Then if the INTR timing is too long for an FERR# signal triggered by the first no-wait instruction to hit the first instruction's interrupt window, it could catch the interrupt window of the second.

The possible malfunction of a no-wait FPU instruction explained above cannot happen if the instruction is being used in the manner for which Intel originally designed it. The no-wait instructions were intended to be used inside the FPU exception handler, to allow manipulation of the FPU before the error condition is cleared, without hanging the processor because of the FPU error condition, and without the need to assert IGNNE#. They will perform this function correctly, since before the error condition is cleared, the assertion of FERR# that caused the FPU error handler to be invoked is still active. Thus the logic that would assert FERR# briefly at a no-wait instruction causes no change since FERR# is already asserted. The no-wait instructions may also be used without problem in the handler after the error condition is cleared, since now they will not cause FERR# to be asserted at all.

If a no-wait instruction is used outside of the FPU exception handler, it may malfunction as explained above, depending on the details of the hardware interface implementation and which particular processor is involved. The actual interrupt inside the window in the no-wait instruction may be blocked by surrounding it with the instructions: PUSHFD, CLI, no-wait, then POPFD. (CLI blocks interrupts, and the push and pop of flags preserves and restores the original value of the interrupt flag.) However, if FERR# was triggered by the no-wait, its latched value and the PIC response will still be in effect. Further code can be used to check for and correct such a condition, if needed. Section D.3.6., "Considerations When FPU Shared Between Tasks", discusses an important example of this type of problem and gives a solution.

D.2.2. MS-DOS* Compatibility Mode in the Pentium® Pro Processor

When bit NE=0 in CR0, the MS-DOS compatibility mode of the Pentium Pro processor provides FERR# and IGNNE# functionality that is almost identical to the Intel486 and Pentium processors. The same external hardware described in Section D.2.1.2., "Recommended External Hardware to Support the MS-DOS* Compatibility Mode", is recommended for the Pentium Pro processor as well as the two previous generations. The only change to MS-DOS compatibility FPU exception handling with the Pentium Pro processor is that all exceptions for all FPU instructions cause immediate error reporting. That is, FERR# is asserted as soon as the FPU detects an unmasked exception; there are no cases in which error reporting is deferred to the next FPU or WAIT instruction. (As is discussed in Section D.2.1.1., "Basic Rules: When FERR# Is Generated", most exception cases in the Intel486 and Pentium processors are of the deferred type.)

Although FERR# is asserted immediately upon detection of an unmasked FPU error, this certainly does not mean that the requested interrupt will always be serviced before the next instruction in the code sequence is executed. To begin with, the Pentium Pro processor executes several instructions simultaneously. There also will be a delay, which depends on the external hardware implementation, between the FERR# assertion from the processor and the responding INTR assertion to the processor. Further, the interrupt request to the PICs (IRQ13) may be temporarily blocked by the operating system, or delayed by higher priority interrupts, and processor response to INTR itself is blocked if the operating system has cleared the IF bit in EFLAGS.

However, just as with the Intel486 and Pentium processors, if the IGNNE# input is inactive, a floating point exception which occurred in the previous FPU instruction and is unmasked causes the processor to freeze immediately when encountering the next WAIT or FPU instruction (except for no-wait instructions). This means that if the FPU exception handler has not already been invoked due to the earlier exception (and therefore, the handler not has cleared that exception state from the FPU), the processor is forced to wait for the handler to be invoked and handle the exception, before the processor can execute another WAIT or FPU instruction.

As explained in Section D.2.1.3., “No-Wait FPU Instructions Can Get FPU Interrupt in Window”, if a no-wait instruction is used outside of the FPU exception handler, in the Intel486 and Pentium processors, it may accept an unmasked exception from a previous FPU instruction which happens to fall within the external interrupt sampling window that is opened near the beginning of execution of all FPU instructions. This will not happen in the Pentium Pro processor, because this sampling window has been removed from the no-wait group of FPU instructions.

D.3. RECOMMENDED PROTOCOL FOR MS-DOS* COMPATIBILITY HANDLERS

The activities of numeric programs can be split into two major areas: program control and arithmetic. The program control part performs activities such as deciding what functions to perform, calculating addresses of numeric operands, and loop control. The arithmetic part simply adds, subtracts, multiplies, and performs other operations on the numeric operands. The processor is designed to handle these two parts separately and efficiently. An FPU exception handler, if a system chooses to implement one, is often one of the most complicated parts of the program control code.

D.3.1. Floating-Point Exceptions and Their Defaults

The FPU can recognize six classes of floating-point exception conditions while executing floating-point instructions:

1. #I — Invalid operation
   #IS — Stack fault
   #IA — IEEE standard invalid operation

2. #Z — Divide-by-zero
3. #D — Denormalized operand
4. #O — Numeric overflow
5. #U — Numeric underflow
6. #P — Inexact result (precision)

For complete details on these exceptions and their defaults, see Section 7.7., “Floating-Point Exception Handling” and Section 7.8., “Floating-Point Exception Conditions”.

D.3.2. Two Options for Handling Numeric Exceptions

Depending on options determined by the software system designer, the processor takes one of two possible courses of action when a numeric exception occurs:

- The FPU can handle selected exceptions itself, producing a default fix-up that is reasonable in most situations. This allows the numeric program execution to continue undisturbed. Programs can mask individual exception types to indicate that the FPU should generate this safe, reasonable result whenever the exception occurs. The default exception fix-up activity is treated by the FPU as part of the instruction causing the exception; no external indication of the exception is given (except that the instruction takes longer to execute when it handles a masked exception.) When masked exceptions are detected, a flag is set in the numeric status register, but no information is preserved regarding where or when it was set.
- Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. The exception handler can then implement any sort of recovery procedures desired for any numeric exception detectable by the FPU.

D.3.2.1. AUTOMATIC EXCEPTION HANDLING: USING MASKED EXCEPTIONS

Each of the six exception conditions described above has a corresponding flag bit in the FPU status word and a mask bit in the FPU control word. If an exception is masked (the corresponding mask bit in the control word = 1), the processor takes an appropriate default action and continues with the computation. The processor has a default fix-up activity for every possible exception condition it may encounter. These masked-exception responses are designed to be safe and are generally acceptable for most numeric applications.

For example, if the Inexact result (Precision) exception is masked, the system can specify whether the FPU should handle a result that cannot be represented exactly by one of four modes of rounding: rounding it normally, chopping it toward zero, always rounding it up, or always down. If the Underflow exception is masked, the FPU will store a number that is too small to be represented in normalized form as a denormal (or zero if it’s smaller than the smallest denormal). Note that when exceptions are masked, the FPU may detect multiple exceptions in a

single instruction, because it continues executing the instruction after performing its masked response. For example, the FPU could detect a denormalized operand, perform its masked response to this exception, and then detect an underflow.

As an example of how even severe exceptions can be handled safely and automatically using the default exception responses, consider a calculation of the parallel resistance of several values using only the standard formula (see Figure D-4). If R1 becomes zero, the circuit resistance becomes zero. With the divide-by-zero and precision exceptions masked, the processor will produce the correct result. FDIV of R1 into 1 gives infinity, and then FDIV of (infinity +R2 +R3) into 1 gives zero.

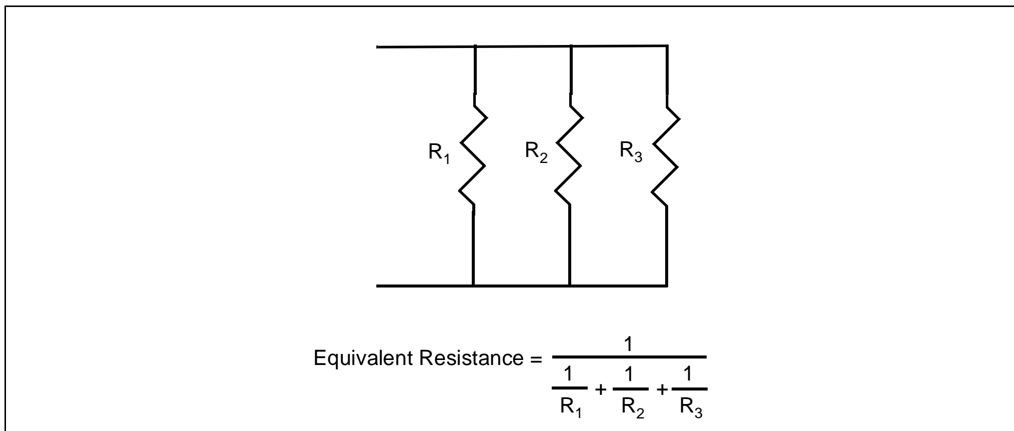


Figure D-4. Arithmetic Example Using Infinity

By masking or unmasking specific numeric exceptions in the FPU control word, programmers can delegate responsibility for most exceptions to the processor, reserving the most severe exceptions for programmed exception handlers. Exception-handling software is often difficult to write, and the masked responses have been tailored to deliver the most reasonable result for each condition. For the majority of applications, masking all exceptions yields satisfactory results with the least programming effort. Certain exceptions can usefully be left unmasked during the debugging phase of software development, and then masked when the clean software is actually run. An invalid-operation exception for example, typically indicates a program error that must be corrected.

The exception flags in the FPU status word provide a cumulative record of exceptions that have occurred since these flags were last cleared. Once set, these flags can be cleared only by executing the FCLEX/FNCLEX (clear exceptions) instruction, by reinitializing the FPU with FINIT/FNINIT or FSAVE/FNSAVE, or by overwriting the flags with an FRSTOR or FLDENV instruction. This allows a programmer to mask all exceptions, run a calculation, and then inspect the status word to see if any exceptions were detected at any point in the calculation.

D.3.2.2. SOFTWARE EXCEPTION HANDLING

If the FPU in or with an Intel Architecture processor (Intel 286 and onwards) encounters an unmasked exception condition, with the system operated in the MS-DOS compatibility mode and with IGNNE# not asserted, a software exception handler is invoked through a PIC and the processor's INTR pin. The FERR# (or ERROR#) output from the FPU that begins the process of invoking the exception handler may occur when the error condition is first detected, or when the processor encounters the next WAIT or FPU instruction. Which of these two cases occurs depends on the processor generation and also on which exception and which FPU instruction triggered it, as discussed earlier in Section D.1., "Origin of the MS-DOS* Compatibility Mode for Handling FPU Exceptions" and Section D.2., "Implementation of the MS-DOS* Compatibility Mode In the Intel486™, Pentium®, and Pentium Pro Processors". The elapsed time between the initial error signal and the invocation of the FPU exception handler depends of course on the external hardware interface, and also on whether the external interrupt for FPU errors is enabled. But the architecture ensures that the handler will be invoked before execution of the next WAIT or floating-point instruction since an unmasked floating-point exception causes the processor to freeze just before executing such an instruction (unless the IGNNE# input is active, or it is a no-wait FPU instruction).

The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# (or ERROR#) output of the processor (or coprocessor), usually through IRQ13 on the "slave" PIC, and then through INTR. Then the external interrupt invokes the exception handling routine. Note that if the external interrupt for FPU errors is disabled when the processor executes an FPU instruction, the processor will freeze until some other (enabled) interrupt occurs if an unmasked FPU exception condition is in effect. If NE = 0 but the IGNNE# input is active, the processor disregards the exception and continues. Error reporting via an external interrupt is supported for MS-DOS compatibility. Chapter 17, *Intel Architecture Compatibility* of the *Intel Architecture Software Developer's Manual, Volume 3*, contains further discussion of compatibility issues.

The references above to the ERROR# output from the FPU apply to the Intel 387 and Intel 287 math coprocessors (NPX chips). If one of these coprocessors encounters an unmasked exception condition, it signals the exception to the Intel 286 or Intel386 processor using the ERROR# status line between the processor and the coprocessor. See Section D.1., "Origin of the MS-DOS* Compatibility Mode for Handling FPU Exceptions", in this appendix, and Chapter 17, *Intel Architecture Compatibility*, in the *Intel Architecture Software Developer's Manual, Volume 3* for differences in FPU exception handling.

The exception-handling routine is normally a part of the systems software. The routine must clear (or disable) the active exception flags in the FPU status word before executing any floating point instructions that cannot complete execution when there is a pending floating point exception. Otherwise, the floating point instruction will trigger the FPU interrupt again, and the system will be caught in an endless loop of nested floating point exceptions, and hang. In any event, the routine must clear (or disable) the active exception flags in the FPU status word after handling them, and before IRET(D). Typical exception responses may include:

- Incrementing an exception counter for later display or printing.
- Printing or displaying diagnostic information (e.g., the FPU environment and registers).

- Aborting further execution, or using the exception pointers to build an instruction that will run without exception and executing it.

Applications programmers should consult their operating system's reference manuals for the appropriate system response to numerical exceptions. For systems programmers, some details on writing software exception handlers are provided in Chapter 5, *Interrupt and Exception Handling*, in the *Intel Architecture Software Developer's Manual, Volume 3*, as well as in Section D.3.4., "FPU Exception Handling Examples", in this appendix.

As discussed in Section D.2.1.2., "Recommended External Hardware to Support the MS-DOS* Compatibility Mode", some early FERR# to INTR hardware interface implementations are less robust than the recommended circuit. This is because they depended on the exception handler to clear the FPU exception interrupt request to the PIC (by accessing port 0F0H) **before** the handler causes FERR# to be de-asserted by clearing the exception from the FPU itself. To eliminate the chance of a problem with this early hardware, Intel recommends that FPU exception handlers always access port 0F0H before clearing the error condition from the FPU.

D.3.3. Synchronization Required for Use of FPU Exception Handlers

Concurrency or synchronization management requires a check for exceptions before letting the processor change a value just used by the FPU. It is important to remember that almost any numeric instruction can, under the wrong circumstances, produce a numeric exception.

D.3.3.1. EXCEPTION SYNCHRONIZATION: WHAT, WHY AND WHEN

Exception synchronization means that the exception handler inspects and deals with the exception in the context in which it occurred. If concurrent execution is allowed, the state of the processor when it recognizes the exception is often **not** in the context in which it occurred. The processor may have changed many of its internal registers and be executing a totally different program by the time the exception occurs. If the exception handler cannot recapture the original context, it cannot reliably determine the cause of the exception or to recover successfully from the exception. To handle this situation, the FPU has special registers updated at the start of each numeric instruction to describe the state of the numeric program when the failed instruction was attempted. This provides tools to help the exception handler recapture the original context, but the application code must also be written with synchronization in mind. Overall, exception synchronization must ensure that the FPU and other relevant parts of the context are in a well defined state when the handler is invoked after an unmasked numeric exception occurs.

When the FPU signals an unmasked exception condition, it is requesting help. The fact that the exception was unmasked indicates that further numeric program execution under the arithmetic and programming rules of the FPU will probably yield invalid results. Thus the exception must be handled, and with proper synchronization, or the program will not operate reliably.

For programmers in higher-level languages, all required synchronization is automatically provided by the appropriate compiler. However, for assembly language programmers exception synchronization remains the responsibility of the programmer. It is not uncommon for a programmer to expect that their numeric program will not cause numeric exceptions after it has

been tested and debugged, but in a different system or numeric environment, exceptions may occur regularly nonetheless. An obvious example would be use of the program with some numbers beyond the range for which it was designed and tested. The example in Section D.3.3.2., "Exception Synchronization Examples", shows a more subtle way in which unexpected exceptions can occur.

As described in Section D.3.1., "Floating-Point Exceptions and Their Defaults", depending on options determined by the software system designer, the processor can perform one of two possible courses of action when a numeric exception occurs.

- The FPU can provide a default fix-up for selected numeric exceptions. If the FPU performs its default action for all exceptions, then the need for exception synchronization is not manifest. However, code is often ported to contexts and operating systems for which it was not originally designed. The example below illustrates that it is safest to always consider exception synchronization when designing code that uses the FPU.
- Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. When an FPU exception handler will be invoked, synchronization must always be considered to assure reliable performance.

The following examples illustrate the need to always consider exception synchronization when writing numeric code, even when the code is initially intended for execution with exceptions masked.

D.3.3.2. EXCEPTION SYNCHRONIZATION EXAMPLES

In the following examples, three instructions are shown to load an integer, calculate its square root, then increment the integer. The synchronous execution of the FPU will allow both of these programs to execute correctly, with INC COUNT being executed in parallel in the processor, as long as no exceptions occur on the FILD instruction. However, if the code is later moved to an environment where exceptions are unmasked, the code in the first example will not work correctly:

Incorrect Error Synchronization

```
FILD  COUNT     ; FPU instruction
INC   COUNT     ; integer instruction alters operand
FSQRT           ; subsequent FPU instruction -- error
                ; from previous FPU instruction detected here
```

Proper Error Synchronization

```
FILD   COUNT    ; FPU instruction
FSQRT           ; subsequent FPU instruction -- error from
                ; previous FPU instruction detected here
INC    COUNT    ; integer instruction alters operand
```

In some operating systems supporting the FPU, the numeric register stack is extended to memory. To extend the FPU stack to memory, the invalid exception is unmasked. A push to a full register or pop from an empty register sets SF (Stack Fault flag) and causes an invalid operation exception. The recovery routine for the exception must recognize this situation, fix up the stack, then perform the original operation. The recovery routine will not work correctly in the first example shown in the figure. The problem is that the value of COUNT is incremented before the exception handler is invoked, so that the recovery routine will load an incorrect value of COUNT, causing the program to fail or behave unreliably.

D.3.3.3. PROPER EXCEPTION SYNCHRONIZATION IN GENERAL

As explained in Section D.2.1.2., "Recommended External Hardware to Support the MS-DOS* Compatibility Mode", if the FPU encounters an unmasked exception condition a software exception handler is invoked **before** execution of the **next** WAIT or floating-point instruction. This is because an unmasked floating-point exception causes the processor to freeze immediately before executing such an instruction (unless the IGNNE# input is active, or it is a no-wait FPU instruction). Exactly when the exception handler will be invoked (in the interval between when the exception is detected and the next WAIT or FPU instruction) is dependent on the processor generation, the system, and which FPU instruction and exception is involved.

To be safe in exception synchronization, one should assume the handler will be invoked at the end of the interval. Thus the program should not change any value that might be needed by the handler (such as COUNT in the above example) until **after** the **next** FPU instruction following an FPU instruction that could cause an error. If the program needs to modify such a value before the next FPU instruction (or if the next FPU instruction could also cause an error), then a WAIT instruction should be inserted before the value is modified. This will force the handling of any exception before the value is modified. A WAIT instruction should also be placed after the last floating-point instruction in an application so that any unmasked exceptions will be serviced before the task completes.

D.3.4. FPU Exception Handling Examples

There are many approaches to writing exception handlers. One useful technique is to consider the exception handler procedure as consisting of "prologue," "body," and "epilogue" sections of code.

In the transfer of control to the exception handler due to an INTR, NMI, or SMI, external interrupts have been disabled by hardware. The prologue performs all functions that must be protected from possible interruption by higher-priority sources. Typically, this involves saving registers and transferring diagnostic information from the FPU to memory. When the critical processing has been completed, the prologue may re-enable interrupts to allow higher-priority interrupt handlers to preempt the exception handler. The standard "prologue" not only saves the registers and transfers diagnostic information from the FPU to memory but also clears the floating point exception flags in the status word. Alternatively, when it is not necessary for the handler to be re-entrant, another technique may also be used. In this technique, the exception flags are not cleared in the "prologue" and the body of the handler must not contain any floating point instructions that cannot complete execution when there is a pending floating point excep-

tion. (The no-wait instructions are discussed in Section 7.5.12., "Waiting Vs. Non-waiting Instructions".) Note that the handler must still clear the exception flag(s) before executing the IRET. If the exception handler uses neither of these techniques the system will be caught in an endless loop of nested floating point exceptions, and hang.

The body of the exception handler examines the diagnostic information and makes a response that is necessarily application-dependent. This response may range from halting execution, to displaying a message, to attempting to repair the problem and proceed with normal execution. The epilogue essentially reverses the actions of the prologue, restoring the processor so that normal execution can be resumed. The epilogue must not load an unmasked exception flag into the FPU or another exception will be requested immediately.

The following code examples show the ASM386/486 coding of three skeleton exception handlers, with the save spaces given as correct for 32 bit protected mode. They show how prologues and epilogues can be written for various situations, but the application dependent exception handling body is just indicated by comments showing where it should be placed.

The first two are very similar; their only substantial difference is their choice of instructions to save and restore the FPU. The trade-off here is between the increased diagnostic information provided by FNSAVE and the faster execution of FNSTENV. (Also, after saving the original contents, FNSAVE re-initializes the FPU, while FNSTENV only masks all FPU exceptions.) For applications that are sensitive to interrupt latency or that do not need to examine register contents, FNSTENV reduces the duration of the "critical region," during which the processor does not recognize another interrupt request. (See the Section 7.3.9., "Saving the FPU's State", for a complete description of the FPU save image.)

After the exception handler body, the epilogues prepare the processor to resume execution from the point of interruption (i.e., the instruction following the one that generated the unmasked exception). Notice that the exception flags in the memory image that is loaded into the FPU are cleared to zero prior to reloading (in fact, in these examples, the entire status word image is cleared).

Examples D-1 and D-2 assume that the exception handler itself will not cause an unmasked exception. Where this is a possibility, the general approach shown in Example D-3 can be employed. The basic technique is to save the full FPU state and then to load a new control word in the prologue. Note that considerable care should be taken when designing an exception handler of this type to prevent the handler from being reentered endlessly.

Example D-1. Full-State Exception Handler

```
SAVE_ALL   PROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU STATE IMAGE
    PUSH   EBP
    .
    .
    MOV    EBP, ESP
    SUB    ESP, 108  ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)
```

```
;SAVE FULL FPU STATE, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSAVE [EBP-108]
    PUSH   [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
    POPFD             ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION
;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED STATE IMAGE
    MOV    BYTE PTR [EBP-104], 0H
    FRSTOR [EBP-108]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV    ESP, EBP
    .
    .
    POP    EBP
;
; RETURN TO INTERRUPTED CALCULATION
    IRETD
SAVE_ALL   ENDP
```

Example D-2. Reduced-Latency Exception Handler

```
SAVE_ENVIRONMENTPROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU ENVIRONMENT
    PUSH   EBP
    .
    .
    MOV    EBP, ESP
    SUB    ESP, 28   ; ALLOCATES 28 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE ENVIRONMENT, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSTENV[EBP-28]
    PUSH   [EBP + OFFSET_TO_EFLAGS]  ; COPY OLD EFLAGS TO STACK TOP
    POPFD             ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED ENVIRONMENT IMAGE
    MOV    BYTE PTR [EBP-24], 0H
```

```
    FLDENV [EBP-28]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV    ESP, EBP
    .
    .
    POP    EBP
;
; RETURN TO INTERRUPTED CALCULATION
    IRETD
SAVE_ENVIRONMENT ENDP
```

Example D-3. Reentrant Exception Handler

```
      .
      .
LOCAL_CONTROL DW ?; ASSUME INITIALIZED
      .
      .
REENTRANTPROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU STATE IMAGE
      PUSH   EBP
      .
      .
      MOV    EBP, ESP
      SUB    ESP, 108   ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE
SIZE)

; SAVE STATE, LOAD NEW CONTROL WORD, RESTORE INTERRUPT ENABLE FLAG (IF)
      FNSAVE [EBP-108]
      FLDCW  LOCAL_CONTROL
      PUSH   [EBP + OFFSET_TO_EFLAGS]  ; COPY OLD EFLAGS TO STACK TOP
      POPFD             ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

      .
      .
;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE. AN UNMASKED
EXCEPTION

; GENERATED HERE WILL CAUSE THE EXCEPTION HANDLER TO BE REENTERED.
; IF LOCAL STORAGE IS NEEDED, IT MUST BE ALLOCATED ON THE STACK.
```

```
;
     .
     .
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED STATE IMAGE
     MOV    BYTE PTR [EBP-104], 0H
     FRSTOR [EBP-108]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
      MOV    ESP, EBP
      .
      .
      POP    EBP
;
; RETURN TO POINT OF INTERRUPTION
      IRETD
REENTRANT ENDP
```

D.3.5. Need for Storing State of IGNNE# Circuit If Using FPU and SMM

The recommended circuit (see Figure D-1) for MS-DOS compatibility FPU exception handling for Intel486 processors and beyond contains two flip flops. When the FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. The assertion of IGNNE# may be used by the handler if needed to execute any FPU instruction while ignoring the pending FPU errors. The problem here is that the state of Flip Flop #2 is effectively an additional (but hidden) status bit that can affect processor behavior, and so ideally should be saved upon entering SMM, and restored before resuming to normal operation. If this is not done, and also the SMM code saves the FPU state, AND an FPU error handler is being used which relies on IGNNE# assertion, then (very rarely) the FPU handler will nest inside itself and malfunction. The following example shows how this can happen.

Suppose that the FPU exception handler includes the following sequence:

```
FNSTSW save_sw       ; save the FPU status word
                     ; using a no-wait FPU instruction
OUT    0F0H, AL      ; clears IRQ13 & activates IGNNE#
    . . . .
FLDCW new_cw         ; loads new CW ignoring FPU errors,
                     ; since IGNNE# is assumed active; or any
                     ; other FPU instruction that is not a no-wait
                     ; type will cause the same problem
     . . . .
```

```
FCLEX              ; clear the FPU error conditions & thus turn off
FERR# & reset the IGNNE# FF
```

The problem will only occur if the processor enters SMM between the OUT and the FLDCW instructions. But if that happens, AND the SMM code saves the FPU state using FNSAVE, then the IGNNE# Flip Flop will be cleared (because FNSAVE clears the FPU errors and thus de-asserts FERR#). When the processor returns from SMM it will restore the FPU state with FRSTOR, which will re-assert FERR#, but the IGNNE# Flip Flop will not get set. Then when the FPU error handler executes the FLDCW instruction, the active error condition will cause the processor to re-enter the FPU error handler from the beginning. This may cause the handler to malfunction.

To avoid this problem, Intel recommends two measures:

1. Do not use the FPU for calculations inside SMM code. (The normal power management, and sometimes security, functions provided by SMM have no need for FPU calculations; if they are needed for some special case, use scaling or emulation instead.) This eliminates the need to do FNSAVE/FRSTOR inside SMM code, except when going into a 0 V suspend state (in which, in order to save power, the CPU is turned off completely, requiring its complete state to be saved.)

2. The system should not call upon SMM code to put the processor into 0 V suspend while the processor is running FPU calculations, or just after an interrupt has occurred. Normal power management protocol avoids this by going into power down states only after timed intervals in which no system activity occurs.

D.3.6. Considerations When FPU Shared Between Tasks

The Intel Architecture allows speculative deferral of floating point state swaps on task switches. This feature allows postponing an FPU state swap until an FPU instruction is actually encountered in another task. Since kernel tasks rarely use floating point, and some applications do not use floating point or use it infrequently, the amount of time saved by avoiding unnecessary stores of the floating point state is significant. Speculative deferral of FPU saves does, however, place an extra burden on the kernel in three key ways:

1. The kernel must keep track of which thread owns the FPU, which may be different from the currently executing thread.

2. The kernel must associate any floating point exceptions with the generating task. This requires special handling since floating point exceptions are delivered asynchronous with other system activity.

3. There are conditions under which spurious floating point exception interrupts are generated, which the kernel must recognize and discard.

D.3.6.1. SPECULATIVELY DEFERRING FPU SAVES, GENERAL OVERVIEW

In order to support multi-tasking, each thread in the system needs a save area for the general purpose registers, and each task that is allowed to use floating point needs an FPU save area large enough to hold the entire FPU stack and associated FPU state such as the control word and status word. (See Section 7.3.9., "Saving the FPU's State", for a complete description of the FPU save image.)

On a task switch, the general purpose registers are swapped out to their save area for the suspending thread, and the registers of the resuming thread are loaded. The FPU state does not need to be saved at this point. If the resuming thread does not use the FPU before it is itself suspended, then both a save and a load of the FPU state has been avoided. It is often the case that several threads may be executed without any usage of the FPU.

The processor supports speculative deferral of FPU saves via interrupt 7 "Device Not Available" (DNA), used in conjunction with CR0 bit 3, the "Task Switched" bit (TS). (See "Control Registers" in Chapter 2 of the *Intel Architecture Software Developer's Manual, Volume 3*.) Every task switch via the hardware supported task switching mechanism (see "Task Switching" in Chapter 6 of the *Intel Architecture Software Developer's Manual, Volume 3*) sets TS. Multi-threaded kernels that use software task switching[1] can set the TS bit by reading CR0, ORing a "1" into[2] bit 3, and writing back CR0. Any subsequent floating point instructions (now being executed in a new thread context) will fault via interrupt 7 before execution. This allows a DNA handler to save the old floating point context and reload the FPU state for the current thread. The handler should clear the TS bit before exit using the CLTS instruction. On return from the handler the faulting thread will proceed with its floating point computation.

Some operating systems save the FPU context on every task switch, typically because they also change the linear address space between tasks. The problem and solution discussed in the following sections apply to these operating systems also.

D.3.6.2. TRACKING FPU OWNERSHIP

Since the contents of the FPU may not belong to the currently executing thread, the thread identifier for the last FPU user needs to be tracked separately. This is not complicated; the kernel should simply provide a variable to store the thread identifier of the FPU owner, separate from the variable that stores the identifier for the currently executing thread. This variable is updated in the DNA exception handler, and is used by the DNA exception handler to find the FPU save areas of the old and new threads. A simplified flow for a DNA exception handler is then:

1. Use the "FPU Owner" variable to find the FPU save area of the last thread to use the FPU.

---

1. In a software task switch, the operating system uses a sequence of instructions to save the suspending thread's state and restore the resuming thread's state, instead of the single long non-interruptible task switch operation provided by the Intel Architecture.
2. Although CR0, bit 2, the emulation flag (EM), also causes a DNA exception, **do not** use the EM bit as a surrogate for TS. EM means that no floating point unit is available and that floating point instructions must be emulated. Using EM to trap on task switches is not compatible with the Intel Architecture's MMX™ technology. If the EM flag is set, MMX instructions raise the invalid opcode exception.

2. Save the FPU contents to the old thread's save area, typically using an FNSAVE instruction.
3. Set the FPU Owner variable to the identify the currently executing thread.
4. Reload the FPU contents from the new thread's save area, typically using an FRSTOR instruction.
5. Clear TS using the CLTS instruction and exit the DNA exception handler.

While this flow covers the basic requirements for speculatively deferred FPU state swaps, there are some additional subtleties that need to be handled in a robust implementation.

D.3.6.3. INTERACTION OF FPU STATE SAVES AND FLOATING POINT EXCEPTION ASSOCIATION

Recall these key points from earlier in this document: When considering floating point exceptions across all implementations of the Intel Architecture, and across all floating point instructions, an floating point exception can be initiated from any time during the excepting floating point instruction, up to just before the next floating point instruction. The "next" floating point instruction may be the FNSAVE used to save the FPU state for a task switch. In the case of "no-wait:" instructions such as FNSAVE, the interrupt from a previously excepting instruction (NE=0 case) may arrive just before the no-wait instruction, during, or shortly thereafter with a system dependent delay. Note that this implies that an floating point exception might be registered during the state swap process itself, and the kernel and floating point exception interrupt handler must be prepared for this case.

A simple way to handle the case of exceptions arriving during FPU state swaps is to allow the kernel to be one of the FPU owning threads. A reserved thread identifier is used to indicate kernel ownership of the FPU. During an floating point state swap, the "FPU owner" variable should be set to indicate the kernel as the current owner. At the completion of the state swap, the variable should be set to indicate the new owning thread. The numeric exception handler needs to check the FPU owner and **discard** any numeric exceptions that occur while the kernel is the FPU owner. A more general flow for a DNA exception handler that handles this case is shown in Figure D-5.

Numeric exceptions received while the kernel owns the FPU for a state swap must be discarded in the kernel without being dispatched to a handler. A flow for a numeric exception dispatch routine is shown in Figure D-6.

It may at first glance seem that there is a possibility of floating point exceptions being lost because of exceptions that are discarded during state swaps. This is not the case, as the exception will be re-issued when the floating point state is reloaded. Walking through state swaps both with and without pending numeric exceptions will clarify the operation of these two handlers.

Case #1: FPU State Swap Without Numeric Exception

Assume two threads A and B, both using the floating point unit. Let A be the thread to have most recently executed a floating point instruction, with no pending numeric exceptions. Let B be the currently executing thread. CR0.TS was set when thread A was suspended. When B starts to

execute a floating point instruction the instruction will fault with the DNA exception because TS is set.

At this point the handler is entered, and eventually it finds that the current FPU Owner is not the currently executing thread. To guard the FPU state swap from extraneous numeric exceptions, the FPU Owner is set to be the kernel. The old owner's FPU state is saved with FNSAVE, and the current thread's FPU state is restored with FRSTOR. Before exiting, the FPU owner is set to thread B, and the TS bit is cleared.

On exit, thread B resumes execution of the faulting floating point instruction and continues.

Case #2: FPU State Swap with Discarded Numeric Exception

Again, assume two threads A and B, both using the floating point unit. Let A be the thread to have most recently executed a floating point instruction, but this time let there be a pending numeric exception. Let B be the currently executing thread. When B starts to execute a floating point instruction the instruction will fault with the DNA exception and enter the DNA handler. (If both numeric and DNA exceptions are pending, the DNA exception takes precedence, in order to support handling the numeric exception in its own context.)

DNA Handler Entry

<other handler set up code>

Current Thread same as FPU Owner?

Yes

No

FPU Owner := Kernel

FNSAVE to Old Thread's FP Save Area (may cause numeric exception)

FRSTOR from Current Thread's FP Save Area

<other handler code>

FPU Owner := Current Thread

<handler final clean-up>

CLTS (clears CR0.TS)

Exit DNA Handler

Figure D-5. General Program Flow for DNA Exception Handler

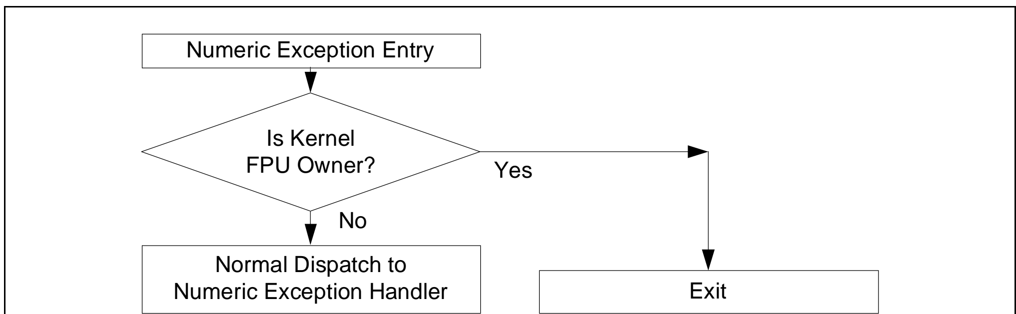


Figure D-6. Program Flow for a Numeric Exception Dispatch Routine

When the FNSAVE starts, it will trigger an interrupt via FERR# because of the pending numeric exception. After some system dependent delay, the numeric exception handler is entered. It may be entered before the FNSAVE starts to execute, or it may be entered shortly after execution of the FNSAVE. Since the FPU Owner is the kernel, the numeric exception handler simply exits, discarding the exception. The DNA handler resumes execution, completing the FNSAVE of the old floating point context of thread A and the FRSTOR of the floating point context for thread B.

Thread A eventually gets an opportunity to handle the exception that was discarded during the task switch. After some time, thread B is suspended, and thread A resumes execution. When thread A starts to execute an floating point instruction, once again the DNA exception handler is entered. B's FPU state is Finessed, and A's FPU state is Frustrate. Note that in restoring the FPU state from A's save area, the pending numeric exception flags are reloaded in to the floating point status word. Now when the DNA exception handler returns, thread A resumes execution of the faulting floating point instruction just long enough to immediately generate a numeric exception, which now gets handled in the normal way. The net result is that the task switch and resulting FPU state swap via the DNA exception handler causes an extra numeric exception which can be safely discarded.

D.3.6.4. INTERRUPT ROUTING FROM THE KERNEL

In MS-DOS, an application that wishes to handle numeric exceptions hooks interrupt 16 by placing its handler address in the interrupt vector table, and exiting via a jump to the previous interrupt 16 handler. Protected mode systems that run MS-DOS programs under a subsystem can emulate this exception delivery mechanism. For example, assume a protected mode O.S. that runs with CR.NE = 1, and that runs MS-DOS programs in a virtual machine subsystem. The MS-DOS program is set up in a virtual machine that provides a virtualized interrupt table. The MS-DOS application hooks interrupt 16 in the virtual machine in the normal way. A numeric exception will trap to the kernel via the real INT 16 residing in the kernel at ring 0. The INT 16 handler in the kernel then locates the correct MS-DOS virtual machine, and reflects the interrupt to the virtual machine monitor. The virtual machine monitor then emulates an interrupt by jumping through the address in the virtualized interrupt table, eventually reaching the application's numeric exception handler.

D.4. DIFFERENCES FOR HANDLERS USING NATIVE MODE

The 8087 has a pin INT which it asserts when an unmasked exception occurs. But there is no interrupt input pin in the 8086 or 8088 dedicated to its attachment, nor an interrupt vector number in the 8086 or 8088 specific for an FPU error assertion. But beginning with the Intel 286 and Intel 287 hardware connections were dedicated to support the FPU exception, and interrupt vector 16 assigned to it.

D.4.1. Origin With the Intel 286 and Intel 287, and Intel386™ and Intel 387 Processors

The Intel 286 and Intel 287, and Intel386 and Intel 387 processor/coprocessor pairs are each provided with ERROR# pins that are recommended to be connected between the processor and FPU. If this is done, when an unmasked FPU exception occurs, the FPU records the exception, and asserts its ERROR# pin. The processor recognizes this active condition of the ERROR# status line immediately before execution of the next WAIT or FPU instruction (except for the no-wait type) in its instruction stream, and branches to the routine at interrupt vector 16. Thus an FPU exception will be handled before any other FPU instruction (after the one causing the error) is executed (except for no-wait instructions, which will be executed without triggering the FPU exception interrupt, but it will remain pending).

Using the dedicated interrupt 16 for FPU exception handling is referred to as the native mode. It is the simplest approach, and the one recommended most highly by Intel.

D.4.2. Changes with Intel486™, Pentium® and Pentium Pro Processors with CR0.NE=1

With these latest three generations of the Intel Architecture, more enhancements and speedup features have been added to the corresponding FPUs. Also, the FPU is now built into the same chip as the processor, which allows further increases in the speed at which the FPU can operate as part of the integrated system. This also means that the native mode of FPU exception handling, selected by setting bit NE of register CR0 to 1, is now entirely internal.

If an unmasked exception occurs during an FPU instruction, the FPU records the exception internally, and triggers the exception handler through interrupt 16 immediately before execution of the next WAIT or FPU instruction (except for no-wait instructions, which will be executed as described in Section D.4.1., “Origin With the Intel 286 and Intel 287, and Intel386™ and Intel 387 Processors”).

An unmasked numerical exception causes the FERR# output to be activated even with NE=1, and at exactly the same point in the program flow as it would have been asserted if NE were zero. However, the system would not connect FERR# to a PIC to generate INTR when operating in the native, internal mode. (If the hardware of a system has FERR# connected to trigger IRQ13 in order to support MS-DOS, but an O/S using the native mode is actually running the system, it is the O/S’s responsibility to make sure that IRQ13 is not enabled in the slave PIC.) With this configuration a system is immune to the problem discussed in Section D.2.1.3., “No-Wait FPU Instructions Can Get FPU Interrupt in Window”, where for Intel486 and Pentium processors a no-wait FPU instruction can get an FPU exception.

D.4.3. Considerations When FPU Shared Between Tasks Using Native Mode

The protocols recommended in Section D.3.6., "Considerations When FPU Shared Between Tasks", for MS-DOS compatibility FPU exception handlers that are shared between tasks may be used without change with the native mode. However, the protocols for a handler written specifically for native mode can be simplified, because the problem of a spurious floating point exception interrupt occurring while the kernel is executing cannot happen in native mode.

The problem as actually found in practical code in a MS-DOS compatibility system happens when the DNA handler uses FNSAVE to switch FPU contexts. If an FPU exception is active, then FNSAVE triggers FERR# briefly, which usually will cause the FPU exception handler to be invoked inside the DNA handler. In native mode, neither FNSAVE nor any other no-wait instructions can trigger interrupt 16. (As discussed above, FERR# gets asserted independent of the value of the NE bit, but when NE=1, the O/S should not enable its path through the PIC.) Another possible (very rare) way a floating point exception interrupt could occur while the kernel is executing is by an FPU immediate exception case having its interrupt delayed by the external hardware until execution has switched to the kernel. This also cannot happen in native mode because there is no delay through external hardware.

Thus the native mode FPU exception handler can omit the test to see if the kernel is the FPU owner, and the DNA handler for a native mode system can omit the step of setting the kernel as the FPU owner at the handler's beginning. Since however these simplifications are minor and save little code, it would be a reasonable and conservative habit (as long as the MS-DOS compatibility mode is widely used) to include these steps in all systems.

Note that the special DP (Dual Processing) mode for Pentium Processors, and also the more general Intel MultiProcessor Specification for systems with multiple Pentium or Pentium Pro processors, support FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatibility mode for systems using more than one processor.

intel®

Index

INDEX

Numerics

16-bit
- address size . . . 3-5
- operand size . . . 3-5

32-bit
- address size . . . 3-5
- operand size . . . 3-5

A

AAA instruction . . . 6-24
AAD instruction . . . 6-24
AAM instruction . . . 6-24
AAS instruction . . . 6-24
AC (alignment check) flag, EFLAGS register . . . 3-13
Access rights, segment descriptor . . . 4-8, 4-11
ADC instruction . . . 6-22
ADD instruction . . . 6-22
Address size attribute
- code segment . . . 3-14
- description of . . . 3-14
- of stack . . . 4-3

Address sizes . . . 3-5
Addressing modes
- assembler . . . 5-9
- base . . . 5-8, 5-9
- base plus displacement . . . 5-9
- base plus index plus displacement . . . 5-9
- base plus index time scale plus displacement . . . 5-9
- displacement . . . 5-8
- effective address . . . 5-8
- immediate operands . . . 5-5
- index . . . 5-8
- index times scale plus displacement . . . 5-9
- memory operands . . . 5-6
- register operands . . . 5-5
- scale factor . . . 5-8
- specifying a segment selector . . . 5-6
- specifying an offset . . . 5-7

Addressing, segments . . . 1-7
Advanced programmable interrupt controller (see APIC)
AF (adjust) flag, EFLAGS register . . . 3-11
AH register . . . 3-7
Alignment
- of words, doublewords, and quadwords . . . 5-1

AND instruction . . . 6-25
APIC, presence of . . . 10-1
Arctangent, FPU operation . . . 7-37
Arithmetic instructions, FPU . . . 7-43
Assembler, addressing modes . . . 5-9
AX register . . . 3-7

B

B (default size) flag, segment descriptor . . . 3-14, 4-3
Base (operand addressing) . . . 5-8, 5-9
Basic execution environment . . . 3-2
B-bit, FPU status word . . . 7-15
BCD . . . 5-4
BCD integers . . . 5-4
- FPU encoding . . . 7-28, 7-29
- packed . . . 5-4, 6-24
- relationship to status flags . . . 3-12
- unpacked . . . 5-4, 6-24

BH register . . . 3-7
Bias value
- numeric overflow . . . 7-51
- numeric underflow . . . 7-52

Biased exponent . . . 7-5
Binary numbers . . . 1-6
Binary-coded decimal (see BCD)
Bit fields . . . 5-4
Bit order . . . 1-5
BOUND instruction . . . 4-15, 6-34, 6-39
BOUND range exceeded exception (#BR) . . . 4-15
BP register . . . 3-7
Branch prediction . . . 2-7
Branching, on FPU condition codes . . . 7-15, 7-36
BSF instruction . . . 6-29
BSR instruction . . . 6-29
BSWAP instruction . . . 6-2, 6-17
BT instruction . . . 3-10, 3-12, 6-29
BTC instruction . . . 3-10, 3-12, 6-29
BTR instruction . . . 3-10, 3-12, 6-29
BTS instruction . . . 3-10, 3-12, 6-29
Bus interface unit . . . 2-8
BX register . . . 3-7
Byte . . . 5-1
Byte order . . . 1-5

C

C1 flag, FPU status word . . 7-13, 7-48, 7-51, 7-53
C2 flag, FPU status word . . . 7-13
Call gate . . . 4-7
CALL instruction . . . 3-14, 4-4, 4-8, 6-31, 6-39
Calls (see Procedure calls)
CBW instruction . . . 6-21
CDQ instruction . . . 6-22
CF (carry) flag, EFLAGS register . . . 3-11
CH register . . . 3-7
CLC instruction . . . 3-12, 6-37
CLD instruction . . . 3-12, 6-37
CLI instruction . . . 6-38, 9-4
CMC instruction . . . 3-12, 6-37
CMOVcc instructions . . . 6-1, 6-16

CMP instruction 6-23
CMPS instruction 3-12, 6-35
CMPXCHG instruction 6-2, 6-18
CMPXCHG8B instruction. 6-2, 6-18, 10-1
Code segment 3-9
Compare
  compare and exchange 6-18
  integers. 6-23
  real numbers, FPU 7-35
  strings. 6-35
Compatibility
  software 1-5
Condition code flags, FPU status word
  branching on. 7-15
  conditional moves on 7-15
  description of 7-12
  interpretation of. .7-14
  use of 7-35
Conditional moves, on FPU condition codes . .7-15
Constants (floating point)
  descriptions of. 7-33
Cosine, FPU operation. 7-37
CPUID instruction. 6-2, 6-40, 10-1, 10-3
CS register. 3-7, 3-9
CTI instruction 6-37
Current privilege level (see CPL)
Current stack 4-2, 4-4
CWD instruction 6-22
CWDE instruction. 6-21
CX register 3-7

D

DAA instruction 6-24
DAS instruction 6-24
Data pointer, FPU 7-21
Data segment. 3-9
Data types
  alignment of words, doublewords, and
    quadwords. 5-1
  BCD integers 5-4, 6-24
  bit fields. 5-4
  byte. 5-1
  doubleword 5-1
  FPU BCD decimal. 7-28
  FPU integer. 7-27
  FPU real number .7-25
  fundamental data types 5-1
  integers. 5-2, 6-22, 6-23
  packed bytes. 8-2
  packed doublewords. 8-2
  packed words 8-2
  pointers. 5-4
  quadword 5-1, 8-2
  strings. 5-4
  unsigned integers 5-4, 6-22, 6-23
  word 5-1
DE (denormal operand exception) flag,
    FPU status word. 7-14, 7-50
DEC instruction. 6-22
Decimal integers, FPU
  description of. 7-28
  encodings 7-29
Denormal number (see Denormalized finite
    number)
Denormal operand exception (#D) 7-50
Denormalization process 7-7
Denormalized finite number 7-6, 7-25
DF (direction) flag, EFLAGS register 3-12
DH register 3-7
DI register. 3-7
Dispatch/execute unit 2-11
Displacement (operand addressing). 5-8, 5-9
DIV instruction 6-23
Division-by-zero exception (#Z) 7-49
Double-extended-precision, IEEE floating-point
    format 7-25
Double-precision, IEEE floating-point
    format 7-25
Double-real floating-point format 7-25
Doubleword 5-1
DS register 3-7, 3-9
DX register 3-7
Dynamic data flow analysis 2-7
Dynamic execution 2-7

E

EAX register 3-5
EBP register 3-5, 4-4, 4-6
EBX register 3-5
ECX register 3-5
EDI register. 3-5
EDX register 3-5
Effective address 5-8
EFLAGS Condition Codes B-1
EFLAGS register 3-10
  restoring from procedure stack 4-7
  saving on a procedure call 4-7
  status flags 7-15, 7-16, 7-35
EIP register. 3-8, 3-14
EMMS instruction 8-9, 8-11
ENTER instruction 4-16, 6-36
ES register 3-7, 3-9
ES (exception summary) flag,
    FPU status word. 7-55
ESC instructions, FPU 7-30
ESI register. 3-5
ESP register 3-5, 4-1, 4-3, 4-4
Exception flags, FPU status word. 7-14
Exception handler. 4-10
Exception priority, FPU exceptions. 7-53
Exception-flag masks, FPU control word 7-17
Exceptions

BOUND range exceeded (#BR) 4-15
description of 4-10
implicit call to handler 4-1
in real-address mode 4-15
notation. 1-7
overflow exception (#OF) 4-15
summary of. 4-12
vector 4-11
Exponent
floating-point number 7-3
Exponential, FPU operation 7-38
Extended real
encodings, unsupported 7-28
floating-point format 7-25

F

F2XM1 instruction 7-38
FABS instruction 7-33
FADD instruction 7-33
FADDP instruction 7-33
Far call
description of 4-4
operation. 4-6
Far pointer
16-bit addressing .3-5
32-bit addressing .3-5
description of 3-3, 5-4
FBSTP instruction 7-32
FCHS instruction 7-33
FCLEX/FNCLEX instructions 7-15
FCMOVcc instructions 6-1, 7-16, 7-32
FCOM instruction 7-15, 7-34
FCOMI instruction 6-1, 7-16, 7-34
FCOMIP instruction 6-1, 7-16, 7-34
FCOMP instruction. 7-15, 7-34
FCOMPP instruction 7-15, 7-34
FCOS instruction 7-13, 7-37
FDIV instruction 7-33
FDIVP instruction 7-33
FDIVR instruction. 7-33
FDIVRP instruction. 7-33
Feature determination, of processor 10-1
Fetch/decode unit. 2-10
FIADD instruction. 7-33
FICOM instruction 7-15, 7-34
FICOMP instruction 7-15, 7-34
FIDIV instruction. 7-33
FIDIVR instruction 7-33
FILD instruction 7-31
FIMUL instruction. 7-33
FINIT/FNINIT instructions . 7-15, 7-16, 7-20, 7-40
FIST instruction 7-31
FISTP instruction 7-32
FISUB instruction 7-33
FISUBR instruction. 7-33
Flat memory model 3-3, 3-8
FLD instruction. 7-31
FLD1 instruction 7-33
FLDCW instruction 7-16, 7-40
FLDENV instruction 7-15, 7-20, 7-24, 7-40
FLDL2E instruction. 7-33
FLDL2T instruction 7-33
FLDLG2 instruction. 7-33
FLDLN2 instruction. 7-33
FLDPI instruction 7-33
FLDSW instruction 7-40
FLDZ instruction 7-33
Floating-point data types 7-24
Floating-point exceptions
automatic handling 7-43
denormal operand exception. 7-50
division-by-zero 7-49
exception conditions 7-47
exception priority 7-53
guidelines for writing exception handlers . . D-1
inexact-result (precision) 7-53
invalid arithmetic operand 7-47, 7-48
MS-DOS compatibility mode D-1
numeric overflow 7-50
numeric underflow. 7-52
software handling 7-45
stack overflow 7-13, 7-47
stack underflow 7-13, 7-47, 7-48
summary of 7-42
synchronization 7-54
Floating-point format
biased exponent 7-5
description of. 7-24
exponent 7-3
fraction 7-3
real number system. 7-2
real numbers 7-25
sign 7-3
significand 7-3
FMUL instruction 7-33
FMULP instruction 7-33
FNOP instruction 7-39
FPATAN instruction 7-37
FPREM instruction 7-13, 7-33, 7-37
FPREM1 instruction 7-13, 7-33, 7-37
FPTAN instruction 7-13
FPU
architecture 7-8
compatibility with Intel Architecture FPUs
and math coprocessors 7-1
floating-point format 7-2, 7-3
IEEE standards 7-1
presence of 10-1
transcendental instruction accuracy 7-39
FPU control word
description of. 7-16
exception-flag masks 7-17
PC field 7-17
RC field 7-18
FPU data pointer 7-21

FPU data registers 7-9
FPU instruction pointer. 7-21
FPU instructions
  arithmetic vs. non-arithmetic instructions . .7-43
  instruction set 7-29
  operands. 7-31
  overview 7-29
  unsupported 7-41
FPU integer
  description of 7-27
  encodings 7-27
FPU last opcode. 7-21
FPU register stack
  description of 7-9
  parameter passing 7-11
FPU state
  image 7-22, 7-23
  saving 7-21
FPU status word
  condition code flags 7-12
  DE flag 7-50
  description of 7-12
  exception flags 7-14
  OE flag 7-50
  PE flag 7-13
  TOP field. 7-9
FPU tag word 7-20
Fraction, floating-point number 7-3
FRNDINT instruction 7-33
FRSTOR instruction. 7-15, 7-20, 7-24, 7-40
FS register 3-7, 3-9
FSAVE/FNSAVE instructions. . . .7-12, 7-15, 7-20,
    7-21, 7-40
FSCALE instruction .7-39
FSIN instruction 7-13, 7-37
FSINCOS instruction 7-13, 7-37
FSQRT instruction 7-33
FST instruction. 7-31
FSTCW/FNSTCW instructions. 7-16, 7-40
FSTENV/FNSTENV
    instructions. 7-12, 7-20, 7-21, 7-40
FSTP instruction. 7-32
FSTSW/FNSTSW instructions 7-12, 7-40
FSUB instruction 7-33
FSUBP instruction 7-33
FSUBR instruction 7-33
FSUBRP instruction .7-33
FTST instruction. 7-15, 7-34
FUCOM instruction. .7-34
FUCOMI instruction 6-1, 7-16, 7-34
FUCOMIP instruction 6-1, 7-16, 7-34
FUCOMP instruction 7-34
FUCOMPP instruction 7-15, 7-34
FXAM instruction 7-13, 7-34
FXCH instruction 7-32
FXTRACT instruction 7-33
FYL2X instruction. 7-38
FYL2XP1 instruction 7-38

G

General-purpose registers 3-5
  parameter passing 4-6
GS register 3-7, 3-9

H

Hexadecimal numbers 1-6

I

ID (identification) flag, EFLAGS register. 3-13
IDIV instruction 6-23
IE (invalid operation exception) flag,
    FPU status word. 7-14, 7-48
IEEE 754 and 854 standards for
    floating-point arithmetic 7-1
IF (interrupt enable) flag, EFLAGS
    register 3-13, 4-11, 9-5
Immediate operands. 5-5
IMUL instruction 6-23
IN 9-3
IN instruction. 6-36, 9-3, 9-4
INC instruction 6-22
Indefinite
  description of. 7-8
  integer 7-27
  packed BCD decimal. 7-28
  real 7-26
Index (operand addressing) 5-8, 5-9
Inexact result, FPU 7-19
Inexact-result (precision) exception (#P) 7-53
Infinity control flag, FPU control word. 7-20
Infinity, floating-point format 7-8
INIT pin 3-10
Input/output (see I/O)
INS instruction 6-36, 9-3, 9-4
Instruction decoder 2-10
Instruction operands 1-6
Instruction pointer (EIP register). 3-14
Instruction pointer, FPU 7-21
Instruction pool (reorder buffer) 2-10
Instruction prefixes (see Prefixes)
Instruction set
  binary arithmetic instructions. 6-22
  bit scan instructions. 6-29
  bit test and modify instructions 6-29
  byte-set-on-condition instructions 6-29
  control transfer instructions 6-30
  data movement instructions 6-16
  decimal arithmetic instructions 6-23
  EFLAGS instructions. 6-37
  floating-point instructions 6-9, 6-11
  integer instructions 6-3
  I/O instructions 6-36
  lists of 6-2
  logical instructions. 6-25
  MMX instructions. 8-4

processor identification instruction 6-40
repeating string operations 6-35
rotate instructions .6-27
segment register instructions 6-38
shift instructions .6-25
software interrupt instructions. 6-34
string operation instructions 6-34
summary. 6-1
system instructions 6-15
test instruction. 6-30
type conversion instructions 6-21
INT instruction 4-15, 6-39
Integers 5-2, 6-22, 6-23
Integer, FPU data type
description of 7-27
indefinite 7-27
Inter-privilege level call
description of 4-7
operation. 4-9
Inter-privilege level return
description of 4-7
operation. 4-9
Interrupt gate 4-11
Interrupt handler. 4-10
Interrupt vector. 4-11
Interrupts
description of 4-10
implicit call to an interrupt handler
procedure 4-11
implicit call to an interrupt handler task. . . .4-14
in real-address mode 4-15
maskable 4-11
summary of. 4-12
user-defined 4-11
vector 4-11
INTn instruction 6-34
INTO instruction 4-15, 6-34, 6-39
Invalid arithmetic operand exception (#IA), FPU
description of 7-48
masked response to 7-49
Invalid operation exception 7-47
INVD instruction 6-2
INVLPG instruction. .6-2
IOPL (I/O privilege level) field, EFLAGS
register 3-13, 9-4
IRET instruction . 3-14, 4-14, 4-15, 6-31, 6-39, 9-5
I/O address space 9-2
I/O instructions
overview of 6-36, 9-3
serialization. 9-6
I/O map base 9-5
I/O permission bit map 9-5
I/O ports
addressing 9-1
defined 9-1
hardware. 9-1
memory-mapped I/O. 9-2
ordering. 9-6
protected mode I/O 9-4
I/O privilege level (see IOPL)
I/O sensitive instructions. 9-4

J

J-bit. 7-3
J*cc* instructions 3-12, 3-14, 6-32
JMP instruction 3-14, 6-30, 6-39

L

L1 (level 1) cache .2-6, 2-8
L2 (level 2) cache .2-6, 2-8
LAHF instruction. 3-10, 6-37
Last instruction opcode, FPU 7-21
LDS instruction 6-39
LEA instruction 6-39
LEAVE instruction. 4-16, 4-21, 6-36
LES instruction 6-39
LGS instruction 6-39
Linear address 3-3
Linear address space
defined 3-3
maximum size 3-3
LOCK signal 6-17
LODS instruction 3-12, 6-35
Log epsilon, FPU operation 7-38
Log (base 2), FPU computation 7-38
Logical address 3-3
LOOP instructions 6-33
LOOPcc instructions. 3-12, 6-33
LSS instruction 6-39

M

Maskable interrupts 4-11
Masked responses
to denormal operand exception. 7-50
to division-by-zero exception. 7-50
to FPU stack overflow or underflow
exception 7-48
to inexact-result (precision) exception. . . . 7-53
to invalid arithmetic operation 7-49
to numeric overflow exception. 7-51
to numeric underflow exception 7-52
Masks, exception-flags, FPU control word . . . 7-17
Memory
order buffer 2-9
organization. .3-2, 3-3
subsystem 2-8
Memory interface unit 2-8
Memory operands. 5-6
Memory-mapped I/O. 9-1, 9-2
MESI (modified, exclusive, shared, invalid) cache
protocol. 2-8
Microarchitecture
detailed description 2-8
overview 2-5
Micro-ops 2-10

MM0, MM1, MM2, MM3, MM4, MM5, MM6,
MM7 registers 8-2
MMX technology
arithmetic instructions 8-8
comparison instructions 8-8
compatibility with FPU architecture. 8-10
conversion instructions 8-9
CPUID instruction .10-2
data transfer instructions 8-6
data types 8-2
detecting MMX technology with CPUID
instruction 8-10
detecting with CPUID instruction 8-11
effect of instruction prefixes on MMX
instructions 8-10
EMMS instruction .8-9
exception handling in MMX code 8-15
instruction operands 8-6
instruction set 8-4, 8-6
interfacing with MMX code 8-12
introduction to 8-1
logical instructions 8-9
memory data formats 8-4
mixing MMX and floating-point
instructions 8-13
programming environment (overview) 8-1
register data formats. 8-4
register mapping .8-15
registers 8-2
saturation arithmetic 8-5
shift instructions 8-9
SIMD execution environment 8-3
support for, determing. 10-2
using MMX code in a multitasking
operating system environment 8-14
using the EMMS instruction 8-11
wraparound mode. 8-5
Modes, operating 3-4
MOV instruction 6-16, 6-38
MOVD instruction 8-6
MOVQ instruction 8-6
MOVS instruction 3-12, 6-35
MOVSX instruction .6-22
MOVZX instruction .6-22
MS-DOS compatibility mode D-1
MTRRs (memory type range registers)
presence of 10-1
MUL instruction 6-23

N

NaN
description of 7-5, 7-8
encoding of 7-6, 7-26
operating on 7-41
SNaNs vs. QNaNs 7-8
Near call
description of 4-4
operation. 4-5
Near pointer
description of. 5-4
Near return
operation 4-5
Near return operation 4-6
NEG instruction 6-23
Non-arithmetic instructions, FPU 7-43
Non-number encodings, FPU 7-5
Non-waiting instructions 7-40, 7-45
NOP instruction 6-40
Normalized finite number 7-4, 7-6
NOT instruction. 6-25
Notation
bit and byte order 1-5
exceptions. 1-7
hexadecimal and binary numbers 1-6
instruction operands 1-6
reserved bits 1-5
segmented addressing 1-7
Notational conventions 1-5
NT (nested task) flag, EFLAGS register 3-13
Numeric overflow exception (#O) 7-13, 7-50
Numeric underflow exception (#U) 7-13, 7-52

O

OE (numeric overflow exception) flag,
FPU status word. 7-14, 7-51
OF (overflow) flag, EFLAGS register . . .3-11, 4-15
Offset (operand addressing). 5-7
Operand
FPU instructions 7-31
instruction 1-6
Operand addressing, modes 5-5
Operand sizes 3-5
Operand-size attribute
code segment 3-14
description of. 3-14
Operating modes 3-4
OR instruction. 6-25
Ordering I/O 9-6
OUT instruction. 6-36, 9-3, 9-4
OUTS instruction 6-36, 9-3, 9-4
Overflow exception (#OF). 4-15
Overflow, FPU exception (see Numeric overflow
exception)
Overflow, FPU stack. 7-47, 7-48

P

Packed BCD integers 5-4
Packed bytes data type 8-2
Packed decimal indefinite. 7-28
Packed doublewords data type 8-2
Packed words data type 8-2
Parameter passing
argument list 4-7
FPU register stack. 7-11
on procedure stack 4-6

on the procedure stack 4-6
through general-purpose registers 4-6
PC (precision) field, FPU control word. 7-17
PE (inexact result exception) flag, FPU status word 7-13, 7-14, 7-19, 7-53
Pentium Pro processor
microarchitecture 2-5, 2-8
PF (parity) flag, EFLAGS register 3-11
Physical address space 3-2
Physical memory 3-2
Pi
description of FPU constant 7-37
Pointers 5-4
POP instruction 4-1, 4-3, 6-20, 6-38
POPA instruction 4-7, 6-20
POPF instruction 3-10, 4-7, 6-37, 9-5
POPFD instruction 3-10, 4-7, 6-37
Privilege levels
description of 4-8
inter-privilege level calls 4-7
stack switching 4-11
Procedure calls
description of 4-4
far call 4-4
for block-structured languages 4-16
inter-privilege level call 4-9
linking 4-3
near call 4-4
overview 4-1
procedure stack 4-1
return instruction pointer (EIP register). 4-4
saving procedure state information. 4-7
stack switching 4-8
to exception handler procedure 4-11
to exception task. 4-14
to interrupt handler procedure 4-11
to interrupt task. .4-14
to other privilege levels. 4-7
types of 4-1
Procedure stack
address-size attribute 4-3
alignment of stack pointer. 4-3
current stack 4-2, 4-4
description of 4-1
EIP register (return instruction pointer). 4-4
maximum size. 4-1
number allowed 4-1
passing parameters on 4-6
popping values from 4-1
procedure linking information 4-3
pushing values on. 4-1
return instruction pointer. 4-4
SS register 4-1
stack pointer 4-1
stack segment. 4-1
stack-frame base pointer, EBP register 4-4
switching. 4-8
top of stack 4-1
width 4-3
Processor identification
earlier Intel architecture processors 10-3
using CPUID instruction 10-1
Processor state information, saving on a procedure call. 4-7
Protected mode
description of. 3-4
I/O 9-4
Pseudo-denormal number 7-29
Pseudo-infinity 7-28
Pseudo-NaN. 7-28
PUSH instruction 4-1, 4-3, 6-19, 6-38
PUSHA instruction 4-7, 6-19
PUSHF instruction 3-10, 4-7, 6-37
PUSHFD instruction 3-10, 4-7, 6-37

Q

QNaN
description of. 7-8
operating on 7-41
rules for generating 7-41
Quadword 5-1, 8-2
Quiet NaN (see QNaN)

R

RC (rounding control) field, FPU control word 7-18
RCL instruction. 6-28
RCR instruction 6-28
RDMSR instruction. 6-2, 10-1
RDPMC instruction. 6-1
RDTSC instruction 6-2, 10-1
Real numbers
encoding 7-5, 7-6, 7-26
floating-point format 7-25
indefinite 7-26
notation 7-5
system. 7-2
Real-address mode 3-4
handling exceptions in. 4-15
handling interrupts in. 4-15
Register operands 5-5
Register stack, FPU 7-9
Registers
EFLAGS register. 3-10
EIP register 3-14
general-purpose registers 3-5
segment registers 3-5, 3-7
Related literature 1-7
REP/REPE/REPZ/REPNE/REPNZ prefixes. 6-35, 9-4
Reserved bits 1-5
RESET pin 3-10
RET instruction. 3-14, 4-4, 6-31, 6-39
Retirement unit 2-12
Return instruction pointer 4-4

Returns, from procedure calls
- exception handler, return from 4-11
- far return 4-6
- interrupt handler, return from 4-11

Returns, from procedures calls
- inter-privilege level return 4-9
- near return 4-5

RF (resume) flag, EFLAGS register 3-13
ROL instruction 6-28
ROR instruction 6-28
Rounding
- control, RC field of FPU control word 7-18
- modes, FPU 7-18
- results, FPU 7-19

RSM instruction 6-2

S

SAHF instruction 3-10, 6-37
SAL instruction 6-25
SAR instruction 6-26
Saturation arithmetic (MMX instructions) 8-5
Saving the FPU state .7-21
SBB instruction 6-22
Scale (operand addressing) 5-8, 5-9
Scale, FPU operation .7-38
Scaling bias value 7-51, 7-52
SCAS instruction 3-12, 6-35
Segment registers
- description of .3-5, 3-7

Segment selector
- description of .3-3, 3-7
- specifying 5-6

Segmented addressing 1-7
Segmented memory model 3-3, 3-8
Segments
- defined 3-3
- maximum number .3-3

Serialization of I/O instructions 9-6
SETcc instructions 3-12, 6-29
SF (sign) flag, EFLAGS register 3-11
SF (stack fault) flag, FPU status word . . 7-15, 7-48
SHL instruction 6-25
SHLD instruction 6-27
SHR instruction 6-25
SHRD instruction 6-27
SI register 3-7
Signaling NaN (see SNaN)
Signed infinity 7-8
Signed zero 7-6
Significand
- of floating-point number 7-3

Sign, floating-point number 7-3
SIMD (single-instruction, multiple-data) execution model 8-3
Sine, FPU operation .7-37
Single-precision, IEEE floating-point format. . .7-25
Single-real floating-point format 7-25
SNaN
- description of 7-8
- operating on 7-41
- typical uses of 7-41

SP register 3-7
Speculative execution 2-6
SS register 3-7, 3-9, 4-1
Stack alignment 4-3
Stack fault, FPU 7-15
Stack overflow and underflow exceptions (#IS), FPU 7-48
Stack overflow exception, FPU 7-13, 7-47
Stack pointer (ESP register) 4-1
Stack segment 3-9
Stack switching
- on calls to interrupt and exception handlers 4-11
- on inter-privilege level calls 4-9, 4-14

Stack underflow exception, FPU 7-13, 7-47
Stack (see Procedure stack)
Stack-frame base pointer, EBP register 4-4
Status flags, EFLAGS register . . 3-11, 7-15, 7-16, 7-35
STC instruction .3-12, 6-37
STD instruction .3-12, 6-37
STI instruction 6-37, 6-38, 9-4
STOS instruction 3-12, 6-35
Strings 5-4
ST(0), top-of-stack register 7-11
SUB instruction 6-22
Superscaler 2-6
Synchronization, of floating-point exceptions . 7-54
System flags, EFLAGS register 3-13
System management mode (SSM) 3-4

T

Tangent, FPU operation 7-37
Task gate 4-14
Task state segment (see TSS)
Tasks
- exception handler 4-14
- interrupt handler 4-14

TEST instruction 6-30
TF (trap) flag, EFLAGS register 3-13
Tiny number 7-7
TOP (stack TOP) field, FPU status word 7-9
Transcendental instruction accuracy 7-39
Trap gate 4-11
TSS
- I/O map base 9-5
- I/O permission bit map 9-5
- saving state of EFLAGS register 3-10

U

UD2 instruction 6-2, 6-40
UE (numeric overflow exception) flag,
FPU status word 7-14, 7-52
Underflow, FPU exception (see Numeric
underflow exception)
Underflow, FPU stack 7-47, 7-48
Underflow, numeric 7-7
Un-normal number 7-28
Unsigned integers 5-4, 6-22, 6-23
Unsupported floating-point formats 7-28
Unsupported FPU instructions 7-41

V

Vector (see Interrupt vector)
VIF (virtual interrupt) flag, EFLAGS register . . 3-13
VIP (virtual interrupt pending) flag, EFLAGS
register 3-13
Virtual 8086 mode
description of 3-13
memory model 3-4
VM (virtual 8086 mode) flag, EFLAGS
register 3-13

W

Waiting instructions 7-40
WAIT/FWAIT instructions 7-40, 7-55
WBINVD instruction 6-2
Word 5-1
Wraparound mode (MMX instructions) 8-5
WRMSR instruction 6-2, 10-1

X

XADD instruction 6-2, 6-18
XCHG instruction 6-17
XLAT/XLATB instruction 6-40
XOR instruction 6-25

Z

ZE (division-by-zero exception) flag,
FPU status word 7-14
Zero, floating-point format 7-6
ZF (zero) flag, EFLAGS register 3-11